

JSF Web应用开发

娄不夜　编著

清华大学出版社

高等学校计算机课程规划教材

JSF Web 应用开发

娄不夜　编著

清 华 大 学 出 版 社

北　京

内 容 简 介

JSF是一种用于构建Java Web应用的标准框架，也是Java EE规范中Web层的标准技术。本书以JSF 2.0为背景，基于JSF 2.0推荐的Facelets视图技术，详细介绍JSF的各项核心技术及其应用。本书同时介绍JPA数据库访问技术，它是Java EE规范中持久层的标准技术。全书共分12章，内容包括Web应用简介、JSF基础、受管bean与EL表达式、使用JSF标记、页面导航、页面布局与数据表格、转换器与验证器、事件处理、资源包与国际化、模板与复合组件、Java DB与实体类、实体管理器与JPQL等。

本书立足基本概念、方法和技术，注重实践与应用环节。对概念、原理和方法的描述力求准确、严谨，对示例力求代码规范、面向实际应用。本书可作为普通高等学校计算机及相关专业的教材，也可作为Web应用开发者学习和使用JSF技术的参考书。

图书在版编目 CIP 数据

JSF Web应用开发/娄不夜编著. —北京：清华大学出版社，2013.4
高等学校计算机课程规划教材
ISBN 978-7-302-30979-6

Ⅰ. ①J…　Ⅱ. ①娄…　Ⅲ. ①网页制作工具—程序设计—高等学校—教材　Ⅳ. ①TP393.092

中国版本图书馆CIP数据核字(2012)第301681号

责任编辑：汪汉友
封面设计：傅瑞学
责任校对：李建庄
责任印制：何　芊

出版发行：清华大学出版社
网　址：http://www.tup.com.cn，http://www.wqbook.com
地　址：北京清华大学学研大厦A座　**邮　编**：100084
社 总 机：010-62770175　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：三河市李旗庄少明印装厂
经　销：全国新华书店
开　本：185mm×260mm　**印　张**：19.5　**字　数**：488千字
版　次：2013年4月第1版　**印　次**：2013年4月第1次印刷
印　数：1～3000
定　价：35.00元

产品编号：043642-01

出 版 说 明

信息时代早已显现其诱人魅力，当前几乎每个人随身都携有多个媒体、信息和通信设备，享受其带来的快乐和便捷。

我国高等教育早已进入大众化教育时代，而且计算机技术发展很快，知识更新速度也在快速增长，社会对计算机专业学生的专业能力要求也在不断翻新。这就使得我国目前的计算机教育面临严峻挑战。我们必须更新教育观念——弱化知识培养目的，强化对学生兴趣的培养，加强培养学生理论学习、快速学习的能力，强调培养学生的实践能力、动手能力、研究能力和创新能力。

教育观念的更新，必然导致教材的更新。一流的计算机人才需要一流的名师指导，而一流的名师需要精品教材的辅助，而精品教材也将有助于催生更多一流名师。名师们在长期的一线教学改革实践中，总结出了一整套面向学生的独特的教法、经验、教学内容等。本套丛书的目的就是推广他们的经验，并促使广大教育工作者进一步更新教育观念。

在教育部相关教学指导委员会专家的帮助和指导下，在各大学计算机院系领导的协助下，清华大学出版社规划并出版了本系列教材，以满足计算机课程群建设和课程教学的需要，并将各重点大学的优势专业学科的教育优势充分发挥出来。

本系列教材行文注重趣味性，立足课程改革和教材创新，广纳全国高校计算机专业一线优秀名师参与，从中精选出佳作予以出版。

本系列教材具有以下特点。

1. 有的放矢

针对计算机专业学生并站在计算机课程群建设、技术市场需求、创新人才培养的高度，规划相关课程群内各门课程的教学关系，以达到教学内容互相衔接、补充、相互贯穿和相互促进的目的。各门课程功能定位明确，并去掉课程中相互重复的部分，使学生既能够掌握这些课程的实质部分，又能节约一些课时，为开设社会需求的新技术课程准备条件。

2. 内容趣味性强

按照教学需求组织教学材料，注重教学内容的趣味性，在培养学习观念、学习兴趣的同时，注重创新教育，加强“创新思维”，“创新能力”的培养、训练；强调实践，案例选题注重实际和兴趣度，大部分课程各模块的内容分为基本、加深和拓宽内容3个层次。

3. 名师精品多

广罗名师参与，对于名师精品，予以重点扶持，教辅、教参、教案、PPT、实验大纲和实验指导等配套齐全，资源丰富。同一门课程，不同名师分出多个版本，方便选用。

4. 一线教师亲力

专家咨询指导，一线教师亲力；内容组织以教学需求为线索；注重理论知识学习，注重学

习能力培养，强调案例分析，注重工程技术能力锻炼。

经济要发展，国力要增强，教育必须先行。教育要靠教师和教材，因此建立一支高水平的教材编写队伍是社会发展的需要，特希望有志于教材建设的教师能够加入到本团队。通过本系列教材的辐射，培养一批热心为读者奉献的编写教师团队。

清华大学出版社

前　　言

JSF(JavaServer Faces)是一种用于构建 Java Web 应用的软件框架。它提供以组件为中心、事件驱动为基础的用户界面构建方法，可以简化和减轻 Java Web 应用的开发难度和开发强度。自 JSF 1.2 随 Java EE 5 于 2006 年发布后，该技术受到了业界的广泛响应和支持。在 Java EE 5 中，JSF 成为 Web 层的三大主要技术(Servlet、JSP 和 JSF)之一。

2009 年 6 月，JSF 2.0 随 Java EE 6 一起发布。在 Java EE 6 中，JSF 已取代 JSP 成为 Web 层的主要技术。与 JSF 1.2 相比，JSF 2.0 不仅支持基于 JSP 的视图技术，也支持 Facelets 视图技术，而且将 Facelets 列为首选的视图语言。David Geary 和 Cay Horstmann 合著的《Core JavaServer Faces(第 3 版)》已经用 Facelets 视图技术取代基于 JSP 的视图技术进行更新，并于 2010 年 5 月出版。

本书以 JSF 2.0 为背景，基于 JSF 2.0 推荐的 Facelets 视图技术，详细介绍 JSF 的各项核心技术及其应用。本书面向的读者要求具有一定的 Java 编程基础，对 HTML 语言最好有一定的了解。本书可以用作普通高等学校相关课程的教学用书，也可供广大 Web 应用开发人员学习和参考之用。

JSF 技术用于构建 Web 应用，本书首先在第 1 章对 Web 及 Web 应用的起源、概念、原理和基本开发过程进行简单介绍，然后在第 2 章对 JSF 的作用、特点及相关概念和原理进行了总体介绍。第 3 章至第 10 章则分别对 JSF 的各单项技术及其应用进行详细介绍。大多数 Web 应用都是以数据库为中心的，本书最后两章介绍 Java 数据库访问技术 JPA 及其在 Web 应用开发中的应用，包括实体类的定义，实体管理以及 Java 持久性查询语言(JPQL)。

全书立足基本理论和方法，注重实践与应用环节。对基本原理、技术和方法的介绍力求概念明确、结构清晰、逻辑严谨。以实际应用为出发点、本章所介绍内容为着眼点，精心设计各章的应用示例。以 MVC 架构为指导思想，介绍各应用示例的开发。各应用示例一般都包括模型、受管 bean 和 JSF 页面几部分。

本书还介绍了一个较为完整的 Web 应用(论坛)的开发，其功能包括登录、注册、浏览主题、发表主题、查看回复、回复主题等。该应用开发的介绍没有独立成章，而是随各章知识点的逐步介绍和推进，分步骤、分层次地展开。

本书采用 JDK 自身携带的 Java DB 为应用所需的数据库管理系统，Oracle 公司官方支持的 NetBeans IDE 为应用开发平台。

本书提供相关的教学资源，包括教学课件以及所有应用示例的源代码。欢迎读者从清华大学出版社网站 http://www.tup.tsinghua.edu.cn 本书相应页面下载和使用。

由于作者学识水平有限，本书难免有错误和不妥之处，敬请广大读者批评指正。如果读者有好的建议或要求，请与作者联系，电子邮箱地址是 loubuye@163.com。

娄不夜

2013 年 2 月

目　　录

第 1 章　Web 应用简介

本章主题：

- 万维网（URL、HTTP、HTML）
- Web 应用的组成、目录结构
- Web 容器、上下文
- Web 应用的生命周期
- NetBeans IDE 操作基础
- Web 应用开发示例

万维网是 Web 应用的载体，Web 应用在很多方面有别于传统的应用软件。本章首先介绍万维网的基础知识，然后介绍 Web 应用的一些基本概念，作为进一步学习 JSF 应用的基础。本章最后介绍了 NetBeans IDE 的一些基本操作，并通过一个简单的例子，演示了 Web 应用的组成及其开发过程中涉及的一些操作。

1.1　Web 基础

Web 是 World Wide Web 的简称，又称 WWW、W3，中文译名为“万维网”，是目前因特网（Internet）上最主要的信息服务形式。万维网由许多互相链接的 HTML 文档等信息资源组成，这些信息资源又由遍布在互联网上的称为 Web 服务器的计算机管理，用户则可以通过客户端浏览器进行浏览。万维网的基本结构如图 1-1 所示。

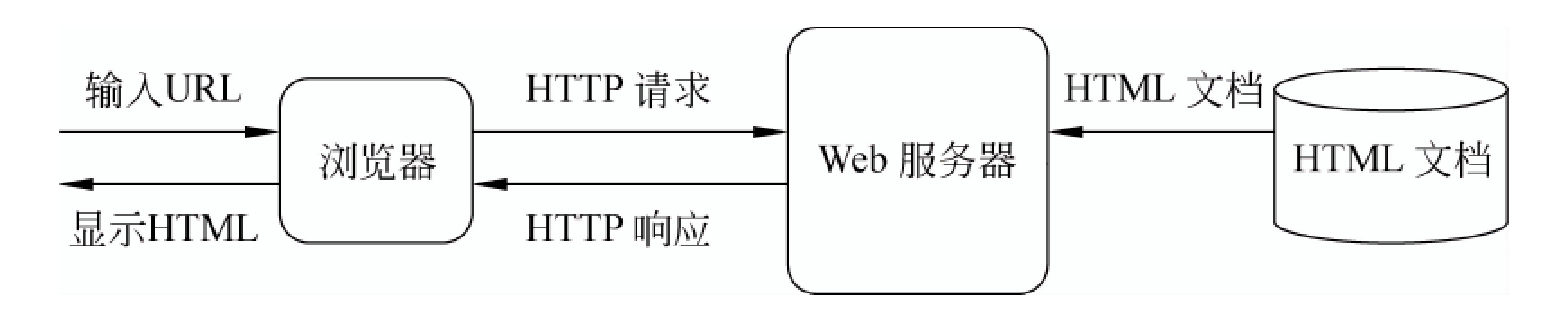


图 1-1　Web 基本结构

万维网技术最早由欧洲核子研究中心的蒂姆·伯纳斯—李（Tim Berners-Lee）于 1990 年提出，其最初的目的是为了解决各个研究项目组和科研人员之间的信息交流和共享问题，避免因信息交流不畅和丢失而造成一些研究工作的重复。万维网技术的核心包括统一资源定位符（URL）、超文本传输协议（HTTP）和超文本标记语言（HTML）。

1.1.1　URL

统一资源定位符（Uniform Resource Locator，URL），也称为网页地址，或简称为网址，是定位因特网上资源的标准地址。URL 的语法格式

<协议类型>://<服务器名>[:<端口号>]/<路径>[?<查询串>]

其中参数说明如下。

(1) 协议类型：指定传输信息所采用的网络协议。最基本的协议是 HTTP，有时也可能会用到其他协议，如 HTTPS、FTP、MAILTO 等。

(2) 服务器名：指定资源所在的 Web 服务器的域名，通常以.com、.net、.org、.gov、.cn 等后缀结尾。有时也可用 IP(Internet Protocal)地址来指定 Web 服务器。

(3) 端口号：端口又指协议端口，是特定应用或进程作为通信端点的软件结构。端口号是一个无符号整数，范围是 0～65 535。对于 Web 服务器提供的 HTTP 通信服务，默认端口号是 80，此时在 URL 中经常省略端口号。

(4) 路径：用于指定资源，通常包括资源在 Web 服务器中的位置信息及名称。路径一般是区分大小写的。

(5) 查询串：是一个经过编码的、由一组名称/值对(用 & 分隔)组成的字符串，表示在 HTTP 请求中发送的数据，也称为请求参数。例如：id=100&p=2。

1.1.2 HTTP

超文本传输协议(HyperText Transfer Protocal，HTTP)是一种建立在 TCP(Transmission Control Protocal)协议之上的属于应用层的网络协议，是万维网上数据通信的基础，适用于分布式、协作式的超媒体信息系统。

1. 请求—响应过程

HTTP 是一种无连接、无状态的协议。无连接并不是指不需要连接，而是指每次连接仅限于一次请求—响应过程。下一次的请求—响应过程需要重新进行连接。无状态是指协议没有记忆约定，前后两次请求—响应过程是相互独立的，协议本身并不会依据上一次请求—响应的状态来处理下一次的请求—响应。

基于 HTTP 协议的请求—响应过程分 4 个步骤。

(1) 建立连接。通过域名(或 IP 地址)，客户端连接到服务器。

(2) 发送请求。建立连接后，客户端把请求信息送到服务器的端口上，完成请求动作。

(3) 发送响应。服务器处理请求，然后向客户端发送响应信息。

(4) 关闭连接。当响应发送完毕，服务器关闭连接。客户端也可以在完整接收响应之前，终止数据传输，关闭连接。

2. 请求信息

客户端向服务器发送的请求信息有较为固定的内容组成和格式，由请求行、请求头和请求体等组成。图 1-2 是请求信息的一个示例。

```
请求行：   POST /hello/hello.html HTTP/1.1
请求头：   Host: localhost:8080
           Accept-Language: zh-cn,zh
           Content-Type: application/x-www-form-urlencoded
           Content-Length: 30
           …
 空行：
请求体：   username=zhang&password=123456
```

图 1-2　请求信息示例

首先是请求行，它包含方法、请求 URI 和协议版本号三项内容，相互之间用空格分隔。

<方法><请求 URI><协议版本号>

方法指定本次请求的性质，即本次请求要对指定的资源做何种操作，如查询、添加、删除、更新等。目前，大多数浏览器仅支持 GET 和 POST 两种方法。一般来说，查询操作应该用 GET 方法，其他操作则可用 POST 方法。通常，采用 GET 方法的请求也称为 GET 请求，采用 POST 方法的请求也称为 POST 请求。

请求 URI(Uniform Resource Identifier)是指 URL 中端口号后面的内容，即不包括其中的协议类型、服务器名和端口号，用于标识资源。这时，有关服务器的信息应该在请求头 Host 域中指定。

请求行的后面是请求头。请求头包含一些域，每个域占一行，由域名和域值组成，两者之间用冒号分隔。有些域可能有多个值。请求头用于指定本次请求以及客户端的一些附加信息。

请求头的后面是一空行(仅包含回车换行符)，该空行表示请求头的结束。

请求信息的最后是可选的请求体。请求体的内容依据请求方法的不同而不同。对 POST 请求，请求体一般仅包含一些请求参数。对 GET 请求，请求体往往是空的，此时请求参数包含在请求行的请求 URI 中。

3. 响应信息

与请求信息一样，服务器向客户端发送的响应信息也有固定的内容组成和格式。响应信息由状态行、响应头和响应体等组成，图 1-3 是响应信息的一个示例。

状态行：	HTTP/1.1 200 OK
响应头：	Server: GlassFish Server Open Source Edition 3.0.1 Content-Type: text/html Content-Length: 232 Date: Mon, 20 Feb 2012 09:55:09 GMT
空行：	…
响应体：	

图 1-3 响应信息示例

响应信息的第一行是状态行，它包含协议版本号、状态码及相应的状态描述信息，相互之间用空格分隔。即

<协议版本号><状态码><状态描述>

状态行中的状态码由 3 位数字组成，其中第一位数字是对响应状态的一个分类。下面是 5 类状态码的一个总体描述。

(1) 1xx：消息，请求已被接收，继续处理。

(2) 2xx：成功，请求已成功被服务器接收、理解、接受并处理。

(3) 3xx：重定向，客户端需要后续操作才能完成这一请求。

(4) 4xx：客户错误，请求含有词法错误或者无法被执行。

(5) 5xx：服务器错误，服务器在处理某个请求时发生错误。

状态行后面是响应头。与请求头类似，响应头也由一些域组成，用于指定本次响应以及服务器的一些附加信息。

响应头的后面是一空行（仅包含回车换行符），该空行表示响应头的结束。

响应信息的最后是响应体，是响应内容本身，如客户请求的 HTML 文档内容。

1.1.3 HTML

超文本标记语言（HyperText Markup Language，HTML）是一种构建 Web 页的主要标记语言。

HTML 文档是一种文本文件，由要显示的内容数据和 HTML 标记组成。HTML 标记用于指定文档的结构、内容的显式格式。每个 HTML 标记由一个小于号、标记名称、属性和一个大于号构成。一般情况下，标记是成对出现的，即以起始标记开始、以结束标记结尾，两者之间的内容是标记的作用范围。

HTML 允许在 Web 页中嵌入图像和对象，也可以包含用于接收用户数据的表单。HTML 允许在页面中嵌入脚本代码（如 JavaScript），来影响页面的行为。HTML 支持 CSS 技术，以便定义页面内容的外观和布局。

Web 浏览器的作用是接收 HTML 文档、解析 HTML 标记，产生人们习惯阅读和浏览的 Web 页面。

1.2 理解 Web 应用

早期的万维网主要由一些 HTML 文档等静态资源组成。随着应用的需要，各种动态网页技术相继出现，如 ASP、Servlet、JSP 等。动态网页的本质是一段程序代码，它可以处理客户请求、并动态产生响应。这样，万维网上不仅包含可供浏览的数据资源，也包含可进行数据处理的软件资源，Web 应用的概念也因此产生。

1.2.1 什么是 Web 应用

Web 应用是指通过 Web 访问的应用软件。一个 Java Web 应用可以由以下元素组成：

- Web 静态资源文档（HTML 页面、图像等）；
- applet（Java 小应用程序）；
- Web 组件（Servlet、JSP 页面、JSF 页面）；
- JavaBean 以及实用类；
- 部署描述符等配置文件。

Web 静态资源和 applet 不在服务器端处理，而是下载到客户端，由客户端解析、显示或运行。Web 组件又称动态页面，当被客户端请求时，在服务器端被解析、处理和运行，并动态产生响应。

JavaBean 以及实用类既可以被 applet 调用，也可以被 Web 组件调用。如果是被 applet 调用，将被下载到客户端运行。如果是被 Web 组件调用，则在服务器端运行。

一个 Web 应用除了上述元素外，往往会包含一个部署描述符（Deployment Descriptor，DD）和其他的一些配置文件。部署描述符等配置文件是一些 XML 文档，使用特定的元素

向服务器描述该 Web 应用的部署要求、所需的服务等信息或参数。其中部署描述符具有特定的文件名，即 web.xml。

许多配置信息不仅可以在部署描述符等配置文件中指定，也可以通过 Java 标注直接出现在 Java 类中。

Web 应用由一些不同性质和作用的元素组成，这些元素被要求按特定的结构组织。首先，所有的元素文件存储在一个被称为 Web 应用文档根的目录下，如图 1-4 所示。

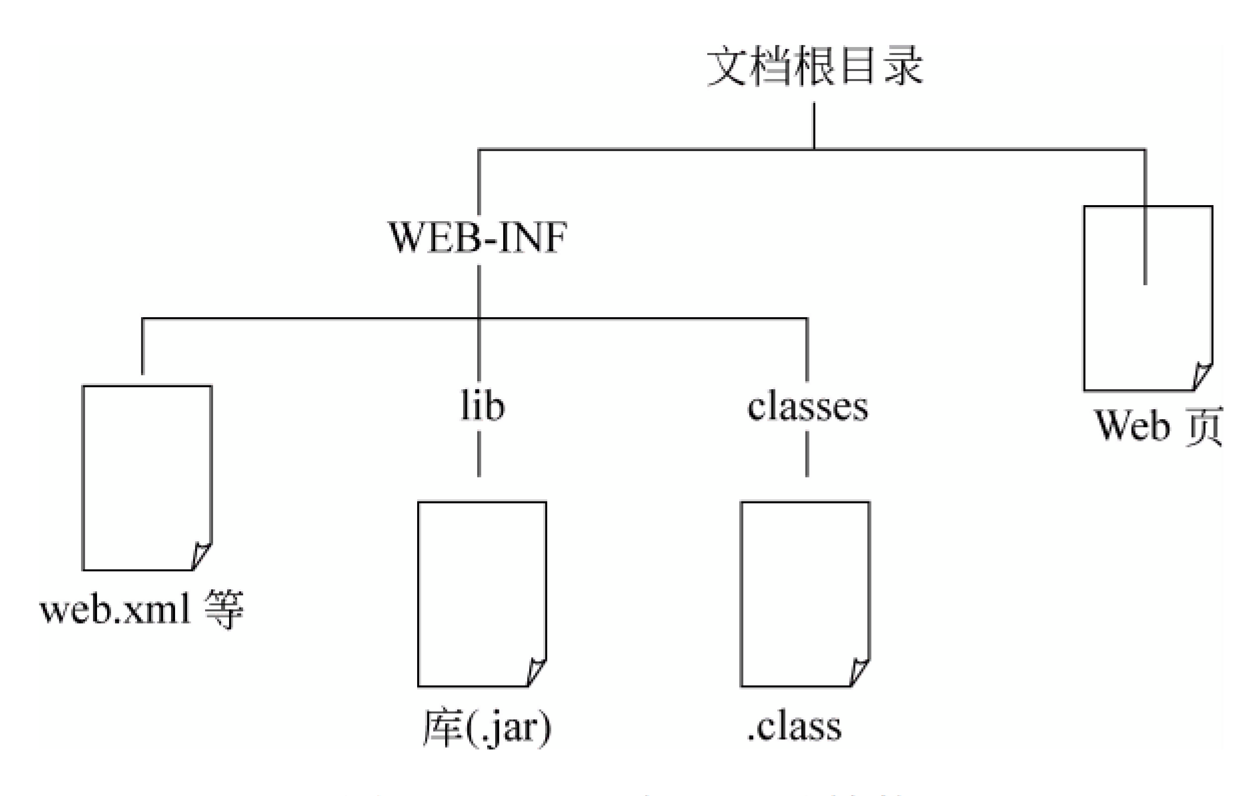


图 1-4　Web 应用目录结构

在这里，文档根目录及其非 WEB-INF 子目录用于存放那些被认为对客户端可见、能被客户端直接访问的文件，包括静态文档、JSP 页、JSF 页、applet 及其相关类。

文档根目录下有一个重要的子目录 WEB-INF，用于存放那些只对 Web 容器和 Web 应用本身可见的文件。其中，classes 子目录通常存放那些属于 Web 应用自身的、在服务器端运行的 Java 类文件，如 servlet 类、JavaBean 类及其他相关的实用类。这些 Java 类可以被组织在不同的包里，所以 classes 子目录下可以有相应的子目录。lib 子目录存放该 Web 应用所需的一些 Java 类库，即 Java 档案文件(Java ARchive file)，也称 JAR 文件。JAR 文件是一种包含一组 Java 类的压缩文件，扩展名为.jar。

部署描述符等配置文件一般直接存放在 WEB-INF 子目录下。

1.2.2　Web 容器

与 Web 静态资源由客户端解析、显示或运行不同，Web 组件在服务器端被解析、处理和运行，并动态产生响应。Web 组件不仅需要在服务器端被进行额外的处理和管理，而且往往需要调用诸如安全性、事务处理等通用性的基础服务。传统的 Web 服务器并不具有这些功能，由此一种被称为 Web 容器的新型软件部件就出现了。

Web 容器是 Web 组件的运行平台，提供诸如请求派发、生命周期管理、安全性、并发性、事务处理、部署等服务。Web 组件可以通过部署描述符等配置文件或 Java 标注配置这些服务，或者利用相应的 API 调用这些服务。

Web 容器的引入降低了 Web 应用的开发难度、提高了 Web 应用的开发效率和可移植性。它使开发人员可以将注意力专注于与特定领域相关的应用问题和数据处理逻辑，而不必纠缠于通用而又复杂的基础服务功能。

Web 组件与静态资源的上述差异，也反映在其请求与响应过程中。图 1-5 是引入 Web

组件和 Web 容器后的 Web 结构示意图。

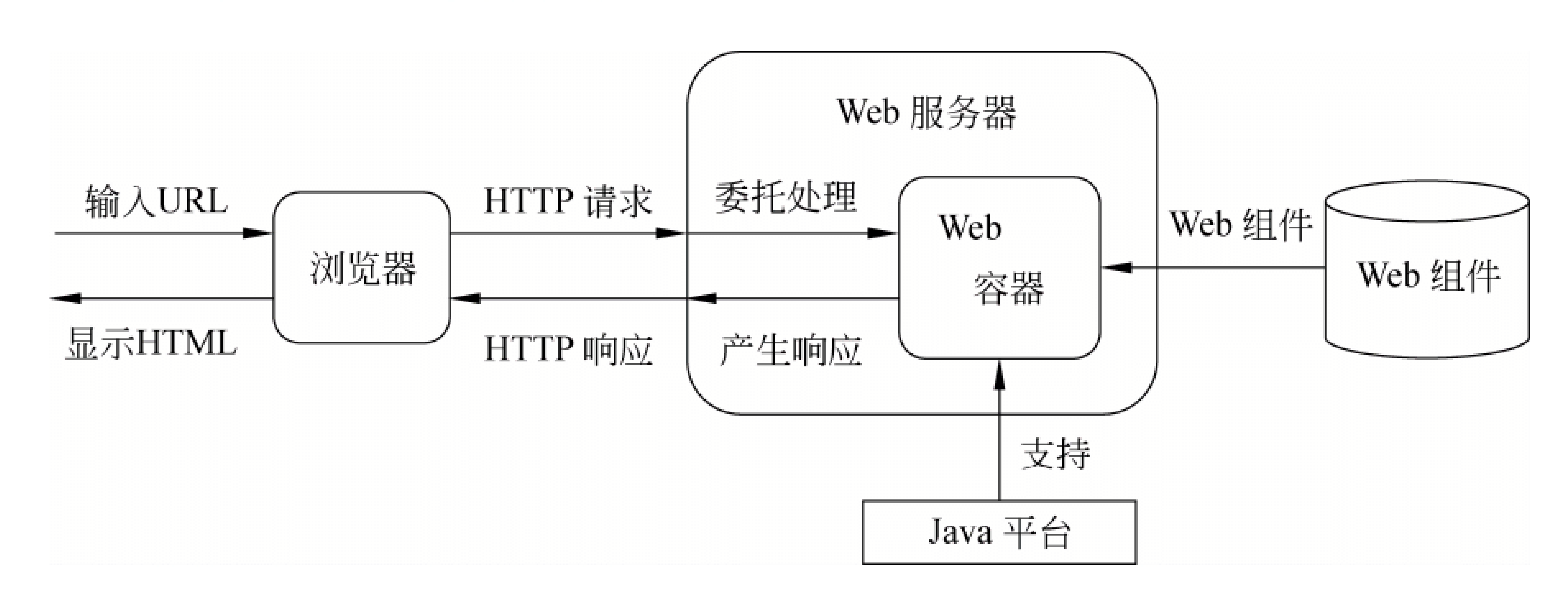


图 1-5　Web 容器及 Web 组件的请求与响应

如果客户请求的是一个静态资源，Web 服务器可以直接读取该资源并产生 HTTP 响应。如果请求的是一个 Web 组件，Web 服务器则将请求委托给 Web 容器处理。Web 容器通过 Web 组件处理请求并产生响应内容，最后由 Web 服务器向客户端发送 HTTP 响应。

以 servlet 组件为例，如果该 servlet 是第一次被客户请求，Web 容器将首先装载该组件相应的类文件，然后实例化并初始化组件的实例对象，最后调用实例对象相应的服务方法产生响应内容。在此期间创建的组件实例对象将被 Web 容器管理。如果该 servlet 组件被再次请求，Web 容器将直接调用已存在的组件实例对象相应的服务方法，产生响应。

1.2.3　Web 应用生命周期

Web 应用有别于传统的软件，一是它由各种具有不同性质和作用的元素组成，而不是单纯地由程序模块(如 Java 类)组成，二是它会通过部署描述符等配置文件或 Java 标注等向 Web 服务器提出运行时的一些要求或所需要的一些服务。因此，Web 应用的开发过程与传统软件的开发过程也会有所不同。下面是创建、部署和运行一个 Web 应用的简要过程。

(1) 开发静态资源、applet、Web 组件、JavaBean 类。

(2) 定义部署描述符及其他的配置文件(可选)。

(3) 编译 applet、servlet、JavaBean 类及其他实用类，并按特定的结构组织各种文件、构建 Web 应用。

(4) 打包产生 Web 档案文件(可选)。

(5) 将 Web 应用部署到 Web 服务器中。

(6) 通过 Web 浏览器访问 Web 应用。

一个 Web 应用的部署描述符及其他配置文件可以根据需要定义，可以有也可以没有。但通常都会有部署描述文件及若干配置文件。

一个 Web 应用可以直接部署到 Web 服务器中，也可以打包成 Web 档案文件后再部署到 Web 服务器中。Web 档案文件(Web ARchive file)又称 WAR 文件，是一种包含 Web 应用的压缩文件，扩展名为. war。WAR 文件和 JAR 文件的压缩机制完全相同，两者的区别主要是：一是文件扩展名不同，二是包含的内容不同。

一个 Web 服务器可以部署多个 Web 应用，每个 Web 应用被部署在一个所谓的上下文

中。每个上下文有一个上下文路径(以/开头、以字符串结尾),客户端应通过上下文路径访问相应 Web 应用中的资源。

1.3　集成开发环境 NetBeans IDE

NetBeans IDE 是一种免费、开源的集成开发环境。它支持开发者利用 Java、C/C++、PHP、JavaScript 和 Groovy 等语言和技术开发专业的桌面应用、企业级应用、Web 应用和移动应用等。

NetBeans IDE 可以从 NetBeans 的官方网站(http://netbeans.org/)上下载。本书使用的 NetBeans IDE 是简体中文、Windows 平台的 NetBeans IDE 6.9.1 完整版,下载的安装程序文件是 netbeans-6.9.1-ml-java-windows.exe。

安装和运行 NetBeans IDE 之前,需要先安装 JDK。通常,不同版本的 NetBeans IDE 需要相应版本的 JDK。比如,NetBeans IDE 6.9.1 需要 JDK 6 Update 13 或之后的版本。

安装好 JDK 后,就可以安装 NetBeans IDE 了。要安装 NetBeans IDE,只需双击运行安装程序文件,然后根据安装向导的提示进行操作即可。默认情况下,安装程序会安装除 Apache Tomcat 之外的所有部件和功能。若有需要,可以在安装欢迎页面中单击“定制”按钮,然后定制需要安装的部件和功能。在安装过程中,向导会引导选择合适的 JDK,在桌面和开始菜单中添加快捷方式(可选)等。

安装好 NetBeans IDE 后,就可以开始工作了。在利用 NetBeans IDE 开发第一个 JSF 应用之前,需要先熟悉一下 NetBeans IDE 界面的基本组成。初始启动 NetBeans IDE 时,其界面主要包含一个显示于编辑窗格内的“起始页”。大多数情况下,NetBeans IDE 界面大致如图 1-6 所示。

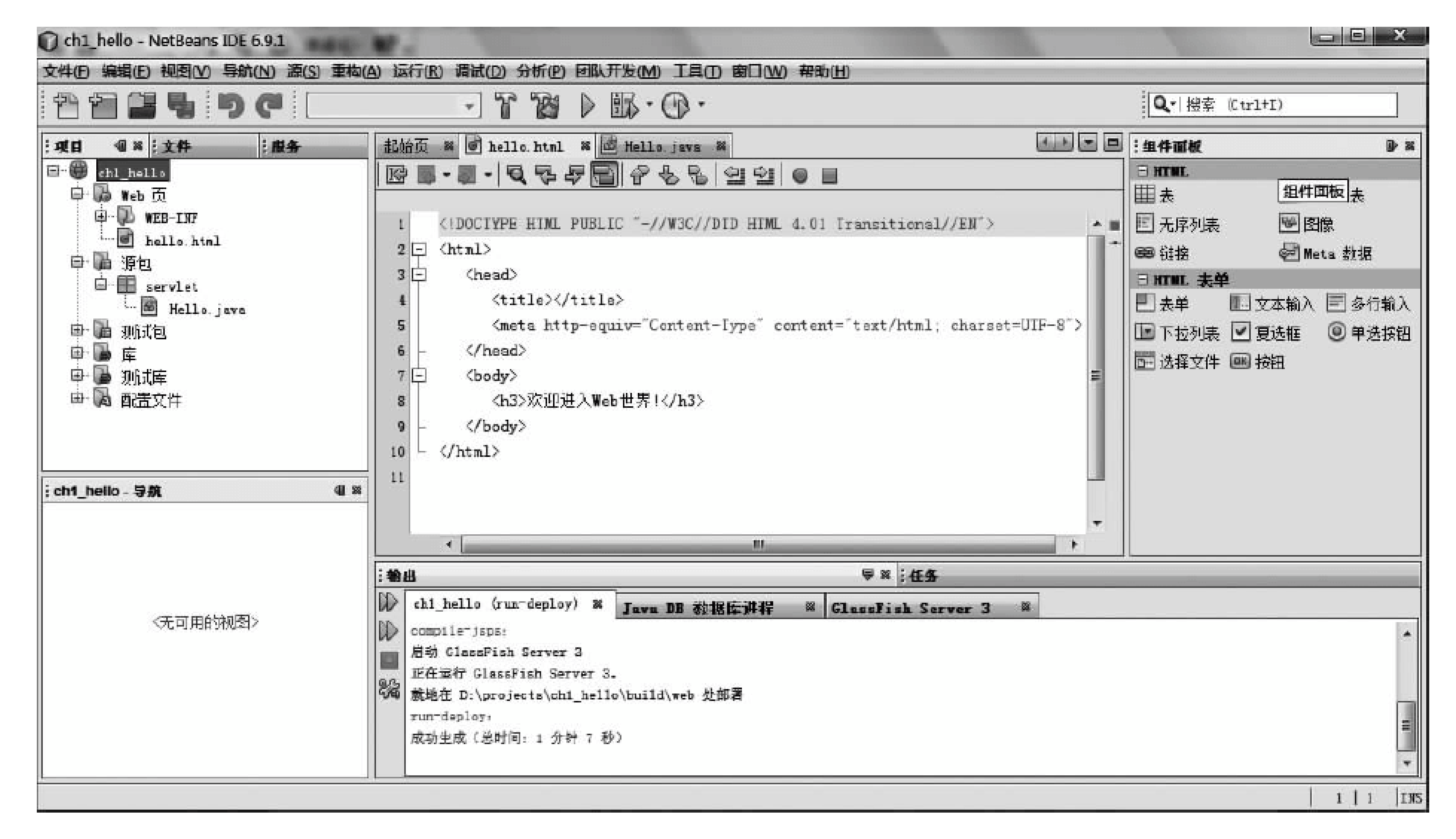


图 1-6　NetBeans IDE 界面

在 NetBeans IDE 界面中,除菜单栏和工具栏外,主要由一些位置和大小可变的窗格组成。其中,编辑器窗格通常位于界面的中间,用于显示和编辑文档内容。每个在编辑器窗格

打开的文档都有一个标签，通过这些标签可以在不同文档间快速切换。其他窗格可以包含一个或多个不同用途的窗口，每个窗口在窗格中也都有一个标签。下面是一些常用的窗口。

1. “项目”窗口

“项目”窗口是项目源的主入口点，显示项目重要内容的逻辑视图。项目可以是 Java 应用程序、EJB 模块、Web 应用程序等。

2. “文件”窗口

“文件”窗口用于显示基于目录的项目视图，其中包括“项目”窗口中未显示的任何文件和文件夹。

3. “服务”窗口

“服务”窗口显示运行时资源的一个逻辑视图，如服务器、数据库等。在这个窗口，可以对这些运行时资源进行一些相关操作，如添加一个服务器，启动或关闭服务器，创建数据库，对数据库执行 SQL 操作等。

4. “导航”窗口

“导航”窗口提供了当前选定文件的简洁视图，并且可以简化文件不同部分之间的导航。在“导航”窗口中双击某个元素，编辑器窗口中的光标就会移至该元素。

5. “输出”窗口

“输出”窗口是一个多标签窗口，可以显示来自 IDE 的各类消息。如对项目编译、打包时产生的消息，启动服务器、部署项目时产生的消息，项目运行时产生的标准输出等。

对各种窗口，用户可以根据需要打开或关闭。要打开一个窗口，可以从“窗口”菜单中选择相应的菜单项或子菜单项。要关闭一个窗口，只需单击该窗口标签右端的“关闭窗口“按钮。某些窗口会在执行相关任务时自动出现。

通过鼠标拖放窗口标签，可以移动窗口。窗口可以在同一窗格内移动，也可以在不同窗格间移动。但不能将窗口移动到编辑器窗格内。编辑器窗格只能包含文档，不能包含其他窗口。反过来，也不能把文档标签移到编辑器窗格之外。

1.4 Web 应用示例

这里，通过一个现有 Web 应用的打开、部署和访问，感性地认识一下 Web 应用的组成及运行原理，并熟悉 NetBeans IDE 的一些基本操作。这个简单 Web 应用的文档根目录为 ch1_hello，其中仅包含两个页面：一个是 html 静态页面，另一个是 servlet 动态页面。

1.4.1 打开并查看 Web 应用

1. 打开 Web 应用

在 NetBeans IDE 中，从“文件”菜单中选择“打开项目”命令，然后在“打开项目”对话框中定位并选择项目文件夹，即 Web 应用的文档根目录(ch1_hello)，最后单击“打开项目”命令按钮打开该 Web 应用。

2. 查看 hello.html

Web 应用打开后，项目窗口中会出现一个相应的项目结点 ch_hello。展开该项目结点下的“Web 页”结点，然后双击 hello.html 文件，可以在编辑器窗格中打开该文件。该页面

非常简单,仅显示一行静态文本。具体代码见代码清单 1-1。

代码清单 1-1　HTML 文档示例

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html;charset=UTF-8">
  </head>
  <body>
    <h3>欢迎进入 Web 世界!</h3>
  </body>
</html>
```

3. 查看 Hello.java

展开该项目结点下的“源包”结点,然后再展开 servlet 子结点,并双击其中的 Hello.java 文件,可以在编辑器窗格中打开该文件。该 servlet 获取当前日期时间,然后利用一个日期格式器将其转换成一个默认格式的文本并输出。具体代码见代码清单 1-2。

代码清单 1-2　servlet 示例

```
package servlet;
import java.io.IOException;
import java.io.PrintWriter;
import java.text.DateFormat;
import java.util.Date;
import javax.servlet.annotation.WebServlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="Hello",urlPatterns={"/Hello"})
public class Hello extends HttpServlet {
  protected void processRequest(HttpServletRequest request,HttpServletResponse
      response)throws ServletException,IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out =response.getWriter();
    try {
      out.println("<html>");
      out.println("<head><title>Servlet</title></head>");
      out.println("<body>");
      Date date =new Date();
      String current =DateFormat.getDateInstance().format(date);
      out.println("<h3>欢迎访问 Servlet</h3>");
      out.println("<h3>当前日期: " +current +"</h3>");
      out.println("</body>");
```

```
      out.println("</html>");
    } finally {
      out.close();
    }
  }

  @Override
  protected void doGet(HttpServletRequest request,HttpServletResponse response)
      throws ServletException,IOException {
    processRequest(request,response);
  }

  @Override
  protected void doPost(HttpServletRequest request,HttpServletResponse response)
      throws ServletException,IOException {
    processRequest(request,response);
  }
}
```

一个 servlet 动态页通常通过扩展 HttpServlet 抽象类定义,其中 doGet 方法用于处理 GET 请求,doPost 方法用于处理 POST 请求。该 servlet 覆盖了这两个方法,使得无论是 GET 请求还是 POST 请求,都将由 processRequest 方法产生响应。

该 servlet 类的@WebServlet 标注指明:该 servlet 的名称是 Hello,url 模式是/Hello。

4. 查看 sun-web.xml

当利用 NetBeans IDE 创建一个 Web 应用,并以 GlassFish 作为服务器时,IDE 会自动产生一个 sun-web.xml 文件。该文件位于"Web 页"结点下的"WEB-INF"子结点中,双击该文件,可以在编辑器窗格中打开它。该文件中的 context-root 元素指定了 Web 应用的上下文路径。

```
<context-root>/ch1_hello</context-root>
```

默认情况下,Web 应用的上下文路径与创建时指定的项目名称是一致的,需要时也可以在此重新指定。

1.4.2 部署和访问 Web 应用

在客户访问 Web 应用前,Web 应用需要先被部署到 Web 服务器中。有些场合,Web 服务器可以根据 Web 应用的文档根目录直接进行部署。而在另外一些场合,一个 Web 应用必须先被打包成 WAR 文件,然后再在 Web 服务器上部署。

1. 打包 Web 应用

要打包本示例的 Web 应用,可以在"项目"窗口中,右击 Web 应用对应的项目结点,即 ch1_hello,然后在打开的快捷菜单中选择"生成"或"清理并生成"命令。

打包产生的 ch1_hello.WAR 文件存放在 Web 应用文档根目录下的 dist 子目录中。在"文件"窗口中可以发现它的存在。

2．部署 Web 应用

要部署本示例的 Web 应用，可以在“项目”窗口中，右击 Web 应用对应的项目结点，即 ch1_hello，然后在打开的快捷菜单中选择“部署”命令。

部署 Web 应用时，如果相应的 Web 服务器还没有启动，NetBeans IDE 会先自动启动服务器。

3．访问 Web 应用

一旦部署了 Web 应用，客户就可以通过浏览器访问它了。上面，已经将示例的 Web 应用部署到了本地主机(localhost)的 GlassFish 服务器(端口号为 8080)上，且其上下文路径为/ch1_hello，因此可以在浏览器地址栏中输入以下 URL 访问相应的页面。

访问静态页面 hello. html 的 URL 为：http://localhost:8080/ch1_hello/hello. html，显示效果如图 1-7 所示。

访问 servlet 动态页面 Hello 的 URL 为：http://localhost:8080/ch1_hello/Hello，显示效果如图 1-8 所示。



图 1-7　请求静态页面 hello. html



图 1-8　请求 servlet 动态页面 Hello

4．取消部署

在 NetBeans IDE 环境下，可以随时查看服务器上部署有哪些 Web 应用，也可以很方便地将某个 Web 应用取消部署。

在“服务”窗口中，依次展开“服务器”、“Glassfish Server 3”和“应用程序”结点，可以查看在 Glassfish 服务器上部署的各个应用程序，其中应该包含本示例 Web 应用对应的 ch1_hello 结点。

要取消本示例 Web 应用的部署，可以右击 ch1_hello 结点，然后从打开的快捷菜单中选择“取消部署”命令。取消部署后，用户就不能对该 Web 应用进行访问了，除非对其重新部署。

1.5　小　　结

- 万维网是因特网上最主要的信息服务形式，也是 Web 应用的载体。
- 万维网技术的核心包括统一资源定位符(URL)、超文本传输协议(HTTP)和超文本标记语言(HTML)。
- Web 应用由一些不同性质和作用的元素组成，这些元素按特定的结构组织，存储在 Web 应用文档根目录下。

- Web 容器是 Web 组件的运行平台，提供诸如请求派发、生命周期管理、安全性、并发性、事务处理、部署等服务。
- 一个 Web 应用部署到 Web 服务器后称为一个上下文，每个上下文有一个上下文路径。
- Web 应用的生命周期大致包括开发静态资源和 Web 组件、定义部署描述符和其他配置文件、编译 Java 类和构建 Web 应用、打包 Web 应用和部署 Web 应用以及访问 Web 应用等几个阶段。

习 题 1

1. 术语解释。
- 万维网；
- URL；
- HTTP；
- HTML；
- Web 应用；
- 上下文与上下文路径。

2. 简述 Web 应用的组成及目录结构。
3. 简述 Web 应用的生命周期。
4. 先下载并解压本章介绍的 Web 应用项目 ch1_hello，然后完成以下操作。

(1) 在 NetBeans 中打开该项目，并查看其中 html 页文件(hello.html)、servlet 页文件(Hello.java)和配置文件(sun-web.xml)的代码。

(2) 部署该 Web 应用，然后通过浏览器请求上述的 html 页和 servlet 页。

(3) 在该 Web 应用项目的 servlet 包中新建一个名为 Calculator 的 servlet 类，将其 URL 模式设为/calculator。该 Servlet 页的功能是计算并输出半径为 1、2、3、4 和 5 的圆的面积和周长，其显示效果如图 1-9 所示。

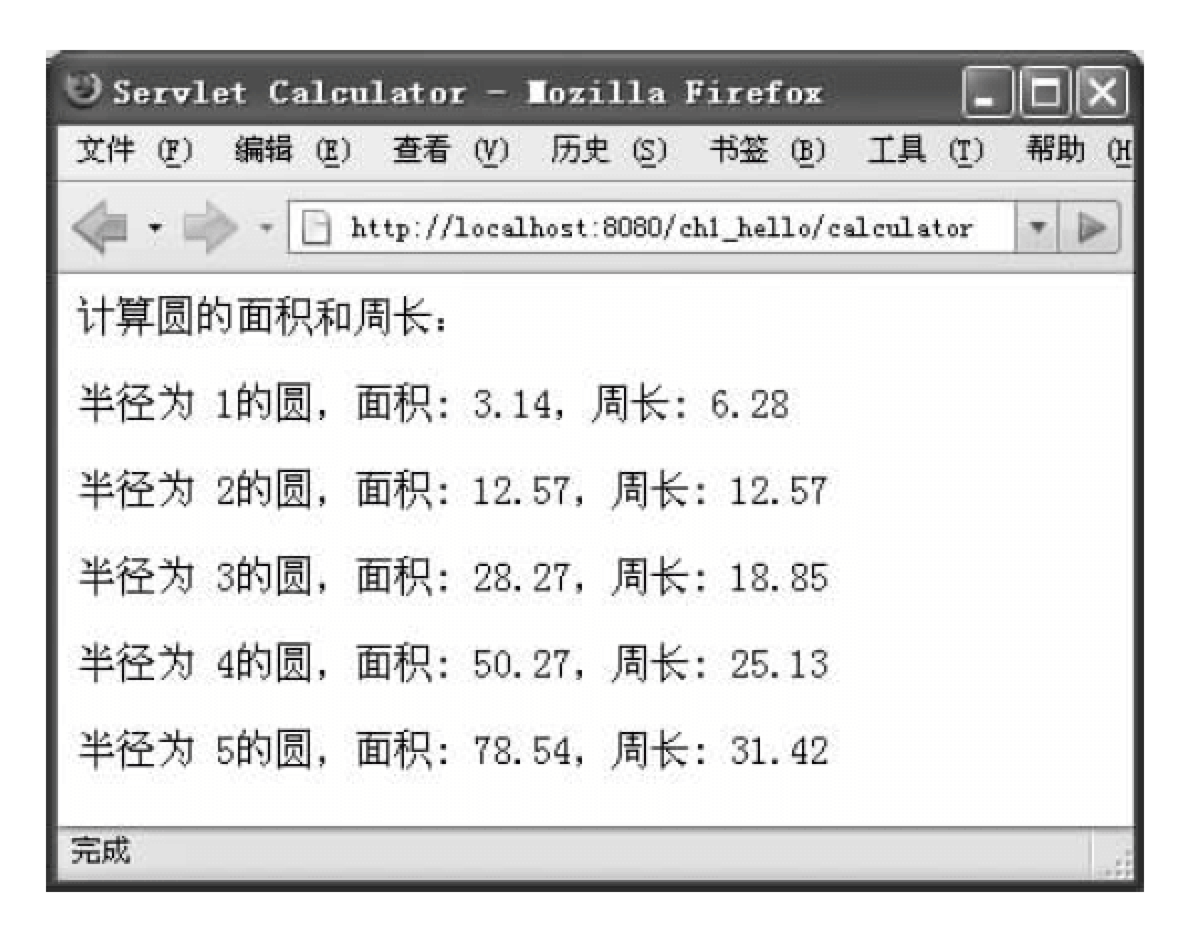


图 1-9 第 4 题示意图

第 2 章　JSF 基础

本章主题：

- 什么是 JSF
- JSF 与 MVC 之间的关系
- JSF 标记与 JSF 组件
- 组件与呈现器
- JSF 请求处理生命周期
- 一个简单的 JSF 应用

了解 JSF 标记、组件、组件树、呈现器等概念，可以更好地理解 JSF 请求处理生命周期。理解了 JSF 请求处理生命周期，有利于熟悉 JSF 的工作原理，从而更加有效地学习 JSF 的各个知识点。理解 MVC 设计架构，能够提供有关 JSF 应用总体结构的一个指导思想。

本章最后通过一个简单的 JSF 应用，介绍 JSF 应用的开发过程，以及 JSF 应用的基本组成。

2.1　JSF 概述

JSF 是 Sun 公司继 JSP 技术之后推出的又一项 Java Web 应用开发技术。JSF 对 Web 应用的一般特性进行了抽象，提出了以组件为中心的事件驱动编程模型，实现了应用表示层和业务层的明确分离，便于 Web 应用的有效开发。

2.1.1　JSF 的定义

JSF(JavaServer Faces)是一种基于 Java 的 Web 应用的用户界面软件框架，它提供了一种以组件为中心、事件驱动的用户界面构建方法，旨在降低 Web 应用的开发难度、减轻开发人员编写和维护 Web 应用的负担。

所谓软件框架是指对某种类型的软件的一种抽象，是该类软件的一个半成品。框架通常定义了这种软件的结构、控制流程，并提供实现一般功能的公共代码。软件开发者可以基于框架，通过覆盖、扩展一些组件来创建特定的应用软件。

JSF 框架作为一个软件半成品，可以由不同的软件厂商提供，但其本身的规范目前由 Oracle 公司控制和维护。最早的 JSF 规范是 JSF 1.0，于 2004 年 3 月发布。2006 年 5 月，推出了通过改进的 JSF 1.2，并随 Java EE 5 一同发布。2009 年 6 月推出的 JSF 2.0，无论在功能还是在性能上都有了很大的提升，是 Java EE 6 标准的一部分。

JSF 框架存在于各种 Java EE 应用服务器，如 GlassFish 服务器。JSF 框架也可以方便地添加到诸如 Apache Tomcat 的轻量级 Web 服务器中。

通常，如果一个 Java Web 应用以 JSF 作为其表示层框架，就称这个应用为 JSF 应用。

2.1.2 JSF 与 MVC 设计架构

MVC 是 Model-View-Controller 的缩写,称为模型—视图—控制器设计架构。MVC 架构将软件分割为松耦合的 3 个部分,各部分各司其职,如图 2-1 所示。

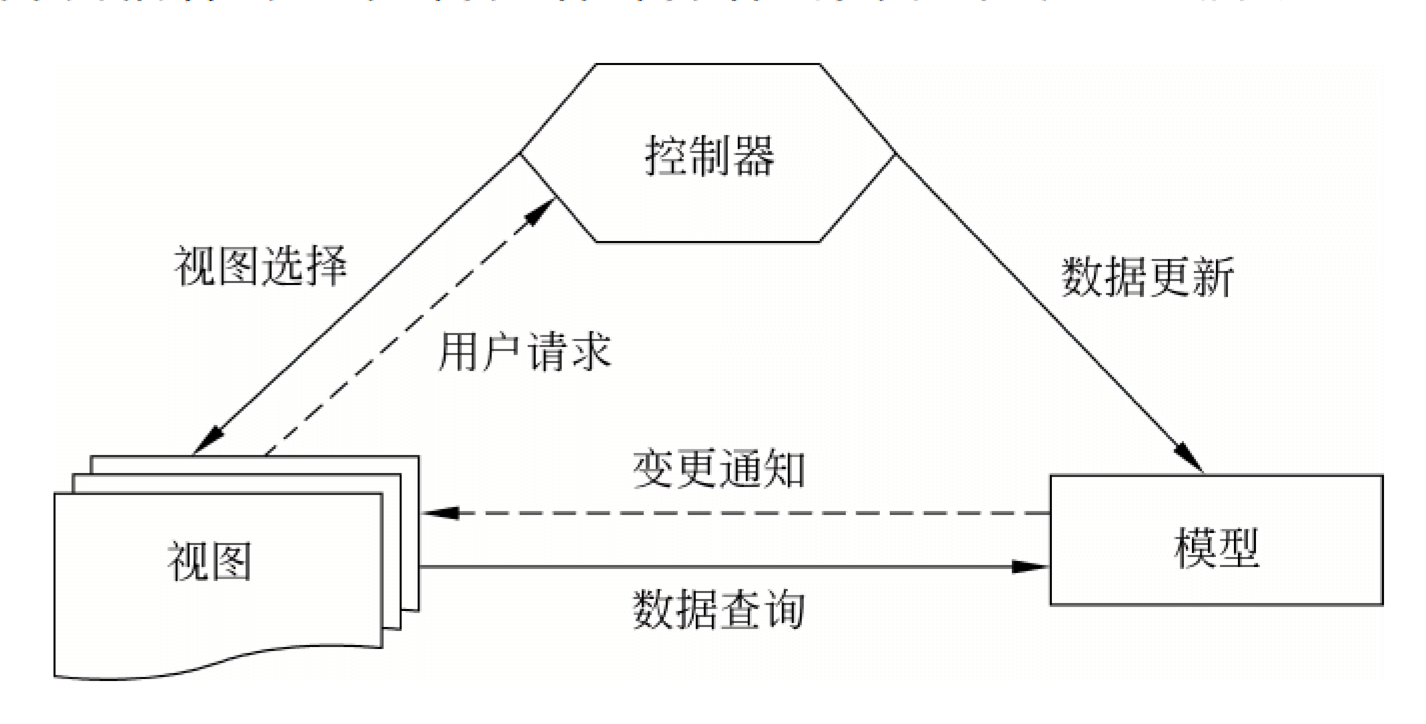


图 2-1 MVC 设计架构

图中实线箭头表示方法或功能调用,虚线箭头表示事件传递或处理。模型封装应用软件的业务逻辑和数据,负责业务逻辑的执行和业务数据的查询和更新,为视图和控制器提供服务。当模型数据变化时,可以通知对此数据感兴趣的视图。视图代表用户交互界面,当扮演“输入”角色时,它允许用户输入数据或引发某种事件;当扮演“输出”角色时,它能从模型获取所需的数据并呈现给用户。控制器负责将用户与视图交互引发的事件转换成模型执行的动作,并决定作为响应的视图的选择。

通常,用作用户交互界面的视图被称为应用的表示层,而实现业务逻辑的模型被称为应用的业务层。

MVC 架构有以下优点。

(1) 利于软件开发。软件各成分按其职责不同被划分为相对独立的 3 个部分,各部分保持松耦合。这为由不同职能的人员、小组相对独立地开发软件的特定部分成为可能。比如,模型组件通常可以独立于控制器和视图、由专门的软件人员单独开发。它降低了软件开发的难度,便于软件的调试、维护和重用。

(2) 一个模型可由多种视图共享。如今,同一个 Web 应用可能需要提供多种用户界面,例如用户希望既能够通过浏览器来收发电子邮件,还能通过手机来访问电子邮箱。这就要求 Web 网站同时能提供 Internet 界面和 WAP 界面,但两种视图应该共享同一模型。

基于 JSF 框架,开发人员可以很好地遵循 MVC 设计架构开发 Web 应用。当然,对于不同类型的软件,架构各部分的实现技术及各部分的通信方式都可能有所不同。在 JSF 应用中,控制器由 JSF 框架中的 FacesServlet、导航处理器等部件以及开发人员提供的事件监听器等组成。视图由开发人员基于 JSF 框架设计开发的 JSF 页面组成。模型可以由普通的 Java 对象、受管 bean 或 EJB 等组成。这里,模型是相对独立的,但通过 EL 表达式、受管 bean 和事件处理机制,可以简单而有效地在视图和模型、控制器和模型之间建立沟通。

2.1.3 JSF 角色

JSF 框架是 JSF 规范的一种实现,JSF 应用是基于 JSF 框架开发的一种 Web 应用。

JSF 规范为 JSF 这个世界定义了 JSF 实现者、工具提供者、页面制作者、组件编写者和应用开发者等 5 种不同角色，它们有各自的职责。

1. JSF 实现者

负责按照 JSF 规范开发具体的 JSF 框架。由于 JSF 规范是 Java EE 标准的一部分，所以实现 Java EE 标准的应用程序服务器也应该包含 JSF 框架，如 GlassFish。

2. 工具提供者

负责提供支持 JSF 应用开发的各种工具，如 NetBeans、Eclipse 等。这些工具可以帮助应用开发者方便地创建组件、设计页面、实施应用的配置管理等，并使开发的 JSF 应用能移植于各种 JSF 框架间。

3. 页面制作者

负责使用 JSF 标记创建页面，实现 JSF 应用的用户界面。页面制作者一般应擅长图形设计、颜色搭配和页面布局，熟悉客户端理解的标记语言（如 HTML）、脚本语言（如 JavaScript），以及产生动态内容的呈现技术（如 JSF 标记），但通常不需要具备丰富的程序设计（如 Java、Visual Basic）能力。

4. 组件编写者

负责开发自定义 UI 组件、呈现器、转换器、验证器以及与用户界面相关的事件处理器，构建可重用的用户界面组件库。JSF 的 UI 组件一般提供以下功能。

- 编码。将组件属性的内部表示转换为呈现页的适当标记。
- 解码。将一个请求的属性（请求参数、头信息、Cookies）转换后存入组件的相应属性中。
- 响应事件并改变组件的外观。
- 支持对用户输入合法性的检验。
- 在请求之间保存和恢复组件状态。

5. 应用开发者

负责开发 JSF 应用的服务器端功能，这些功能与用户界面没有直接的关系。应用开发者一般要担负以下职责。

- 定义持久性存储机制，如设计和创建关系数据库。
- 创建表示持久性数据和实现持久性功能的相关软件，如实体类、受管 bean、EJB 等。
- 创建封装 JSF 应用功能或业务逻辑的相关软件，如普通 Java 类、受管 bean、EJB 等。
- 以适当方式向 JSF 应用的表示层暴露业务数据和业务逻辑。

通常，开发一个 JSF 应用仅涉及后面 3 种角色，即页面制作者、组件编写者和应用开发者。每种角色所需开发人员数量会因应用项目的规模和性质不同而不同，一个开发人员也可能扮演多种角色。

2.2 JSF 组件

JSF 技术的核心是组件技术，它将用户界面元素表示为可重用 Java 组件对象，将 HTTP 请求参数看作为相关组件对象的新状态，从而为在 Web 应用中引入基于组件的事件驱动编程模型奠定了技术基础。

2.2.1 组件与组件标记

JSF 页面文件主要由 JSF 标记组成,每个标记都有相关的标记处理程序。当客户访问一个 JSF 页面时,JSF 框架将读取该 JSF 页面文件,解析各 JSF 标记、执行相应的标记处理程序,并构建相应的组件树。

组件树中的组件是组件类的实例对象,是 JSF 页面文件中的标记在服务器端的内部表示。每个组件都有自己的属性及相关的访问方法。组件树又称为视图,是 JSF 页面在服务器端的内部表示。视图一旦创建便被保存在当前的 FacesContext 对象中。

组件树的根是 UIViewRoot 对象,通过该对象可以访问组件树中的所有组件。下面代码可以获得当前正在处理的视图的组件树的根:

```
UIViewRoot root=FacesContext.getCurrentInstance().getViewRoot();
```

图 2-2 是一个组件树示意图。该图表明,该组件树根结点下有多个组件实例,其中一个 UIOutput 型组件可能对应于 JSF 页面文件中的 h:body 标记,它有一个 HtmlForm 型子组件,该子组件下又包含一个 HtmlOutputLable 型子组件、一个 HtmlInputText 型子组件和一个 HtmlCommandButton 型子组件。

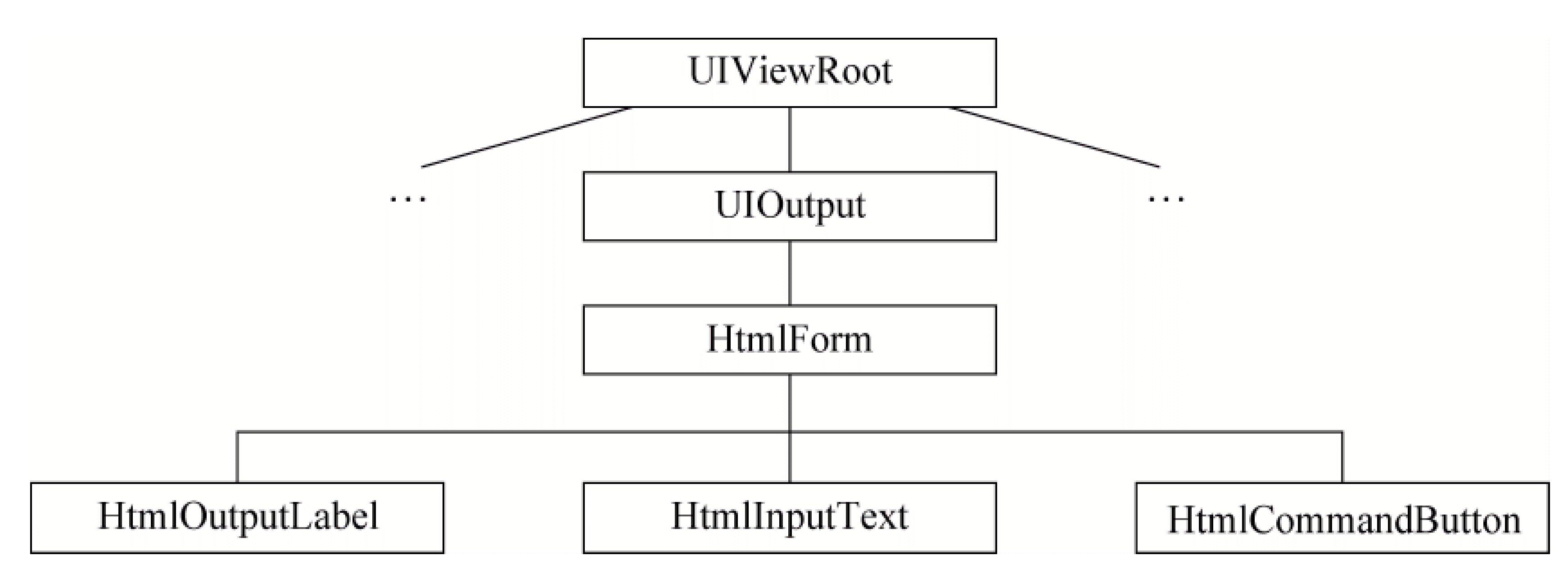


图 2-2 组件树示意图

说明:这里把 JSF 页面在服务器端的内部表示(即组件树)称为视图,但从 MVC 架构的角度来考虑 JSF 应用,JSF 页面本身也经常被称为视图。

2.2.2 呈现器

在服务器内部,一个被客户请求的 JSF 页面被转换成一棵组件树。当响应时,组件树中的组件需要由特定的对象进行处理、产生输出,这种对象称为呈现器(renderer)。整个组件树产生的输出就是服务器对客户请求的响应。

呈现器被组织成呈现包(renderkit),每个呈现包通常用于特定类型的输出,即相同的组件树采用不同的呈现包,可产生不同类型的输出,如 HTML、WML 等。JSF 规范要求,所有的 JSF 实现(即 JSF 框架)必须提供与 HTML4.01 兼容的 HTML 呈现包,以便能对 JSF 页面(视图)进行处理、产生标准的 HTML 文档。

说明:呈现包不是 Java 包,而是一种 Java 对象,用于管理一组呈现器。

呈现器既负责从服务器端组件到客户端响应的转换,也负责对请求参数进行解析并将其保存在服务器端特定的组件内。可以将呈现器想象成服务器和客户机之间的翻译。当

JSF 从客户端接收到一个请求时，呈现器将进行解码(decoding)，即提取与组件相关的请求参数并保存于组件中。当要产生响应时，呈现器将进行编码(encoding)，即创建客户能理解的组件的表示方式。

例如，客户请求的 JSF 页面文件中包含以下标记：

```
<h:inputText id="input1" size="20" maxlength="30"/>
```

当 JSF 框架接受并处理请求时，将创建一棵组件树，组件树中包含与上述标记相对应的 HtmlInputText 型组件。响应时，呈现器将对组件进行编码产生如下 HTML 标记：

```
<input id="form1:input1" name="form1:input1" type="text" maxlength="30" size="20"/>
```

呈现功能既可以由特定呈现器来完成，称为委托实现，也可以由组件自身完成，称为直接实现。

2.2.3 组件标识符和客户端标识符

每个组件可以有一个组件标识符。组件标识符可以在组件标记中用 id 属性指定。如果页面制作者没有为组件指定标识符，需要时 JSF 框架会自动为组件指定标识符。

标识符必须以字母、下划线打头，并且由字母、数字、连字符、下划线组成。对命名容器(NamingContainer)组件(如 HTMLForm、HtmlDataTable)，其所有子组件都应该有唯一的标识符，但不同命名容器组件中子组件的标识符可以相同。在服务器端，可以通过组件的 getId()和 setId(String)访问组件标识符。

一个组件可以有一个客户端标识符。客户端标识符是指组件呈现产生的 HTML 标记的标识符(id 属性)。一般情况下，组件的客户端标识符与其组件标识符相同。

在一棵组件树中，有些组件的标识符是可以相同的，但在一个 HTML 文档中，所有标记的 id 属性都应该是唯一的。为满足这一要求，JSF 框架一般将命名容器组件中的组件的客户端标识符设置为：＜命名容器组件的客户端标识符＞:＜组件自身标识符＞。比如，若表单组件(标识符为 form1)中有一个文本域子组件(标识符为 input1)，则该文本域子组件的客户端标识符将被设置为 form1:input1。

2.3 请求处理生命周期

JSF 是一种 Java Web 应用的表示层框架，其主要内容是对处理 HTTP 请求的过程进行了抽象，并就其总体结构、公共服务等提供了实现。它用组件树(视图)来表示一个 JSF 页面，将请求参数存储为相应组件的状态，将用户单击递交按钮模拟为递交按钮组件引发了动作事件等。JSF 规范定义了处理请求的整个过程——请求处理生命周期。这一过程主要由恢复视图、应用请求值、处理验证、更新模型值、调用应用和呈现响应等 6 个阶段组成，如图 2-3 所示。

图中实线所示是一个正常的控制流，虚线是一些可选的控制流。JSF 处理一个请求时，大多数阶段后，都会将合适的事件广播到相应的事件监听器。事件监听器在处理完事件后可以正常地进入下一阶段，也可以跳过其他阶段直接进入呈现阶段，或跳过其余所有阶段并由自己呈现响应。

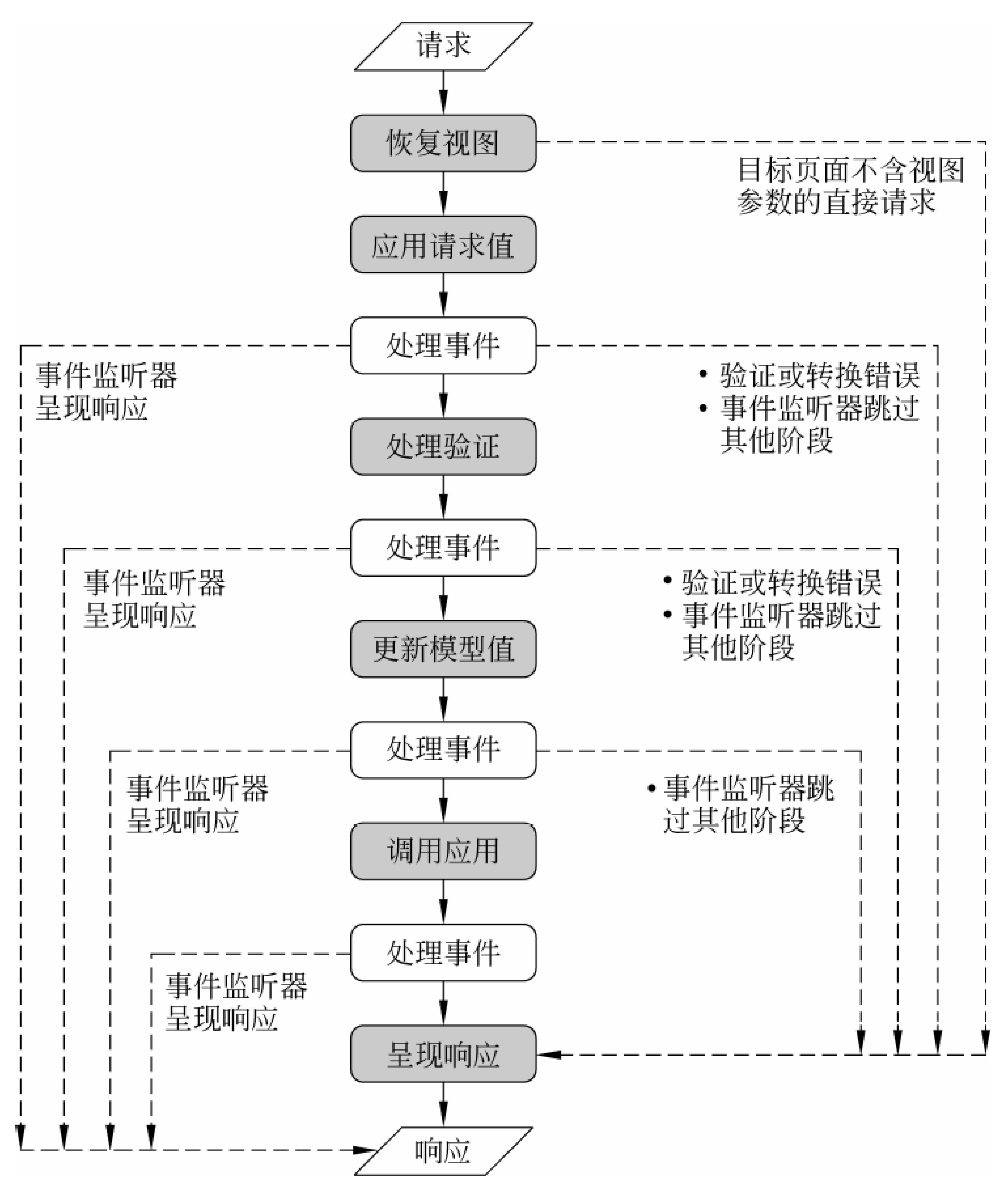


图 2-3　JSF 请求处理生命周期

2.3.1　阶段 1：恢复视图

在 JSF 中，可以将客户对 JSF 页面的请求分为回送(postback)请求和直接请求两大类。回送请求是 POST 请求，而直接请求通常是 GET 请求。

回送请求是指先将请求提交给当前页面自身、然后在服务器端经过导航处理再转至目标页面(也可能是当前页面)的请求。对回送请求，JSF 框架首先会恢复页面视图，然后进入请求处理后续阶段。作为回送请求的结果，浏览器窗口中显示的目标页面与浏览器地址栏中的 URL 通常是不一致的。

直接请求是指发送请求时有明确的目标页面、且指定在请求 URL 中的请求。作为直接请求的结果，浏览器窗口中显示的目标页面与浏览器地址栏中的 URL 通常是一致的。

直接请求又可分为目标页面含视图参数和不含视图参数两种。对目标页面不含视图参数的直接请求，JSF 框架会创建页面视图，然后跳过其他阶段、直接进入呈现响应阶段。对目标页面含视图参数的直接请求，JSF 框架会先创建页面视图，然后进入下一阶段，完成应用请求值、处理验证等相关处理。

视图一旦被创建或恢复，将被保存在当前的 FacesContext 对象中。

2.3.2 阶段 2：应用请求值

首先，JSF 框架会将所有的请求参数保存在一个映射中，映射中每个元素是一个请求参数的名称—值对。然后，每个组件（或其呈现器）从请求参数映射中查找属于自己的请求参数。一旦找到，将根据组件类型进行相应的处理。

- 若为 EditableValueHolder 组件（如 UIInput），参数值被保存在组件中。这个值称为被提交值（submitted value）。
- 若为 ActionSource 组件（如 UICommand），表明该组件被激活，则产生一个动作事件（ActionEvent）放入事件队列。

说明：

（1）通常，对组件值的验证处理将在下一阶段进行，但对直接组件（其 immediate 属性值为 true），组件值的验证处理会在本阶段立即进行，并可能会引发转换或验证错误。

（2）通常，动作事件的处理将在调用应用阶段进行，但对直接组件（其 immediate 属性值为 true）引发的动作事件会在本阶段立即处理，并经导航处理后直接进入呈现响应阶段。

2.3.3 阶段 3：处理验证

在该阶段，JSF 框架遍历组件树并验证每个组件的被提交值是否有效。

- 由相关的标准转换器或自定义转换器对被提交值进行数据类型或格式转换；
- 若组件的 required 属性值为 true，检查已进行转换处理的被提交值是否空；
- 由相关的标准验证器或自定义验证器对已进行转换处理的被提交值进行正确性检查。

如果转换和验证成功，组件的被提交值转换为本地值。此时可能触发值变化事件并被相关的监听器处理，然后转入请求处理生命周期的下一阶段。值变化事件监听器可以跳过其余阶段、转至呈现响应阶段，或直接产生响应。

如果转换或验证失败，将附带转换或验证错误信息进入呈现响应阶段。

2.3.4 阶段 4：更新模型值

进入该阶段时，所有组件的本地值都具有正确的类型，且是有效的。

在该阶段，JSF 框架将基于组件标记中指定的值绑定表达式，用组件的本地值更新支撑 bean（受管 bean）中的相关属性。对于 JSF 框架来说，受管 bean 也被看作是模型的一部分。

2.3.5 阶段 5：调用应用

进入该阶段时，支撑 bean 已被更新。

在该阶段，JSF 框架将向所有已注册的相关监听器广播动作事件，执行监听器方法和动作方法。

监听器方法和动作方法可以调用模型对象的有关方法完成相应的业务处理。业务处理的结果可以保存在适当的支撑 bean 中，以便将它们包含在响应中。

动作方法返回一个结果字符串给导航处理器，导航处理器以此决定作为响应的页面。

2.3.6　阶段 6：呈现响应

进入该阶段时，所有请求参数处理和业务数据处理都已经完成。

本阶段的主要任务是完成编码工作，即由组件树中的各组件(或其呈现器)基于组件状态产生客户能够理解的表示。整个组件树产生的输出作为响应由 Web 服务器送往客户端。若作为响应的页面不同于当前页面，则首先创建响应页面的视图，然后再完成编码工作。

本阶段的另一个任务是保存响应页面视图，以便用户再次请求时可以在恢复视图阶段恢复该视图。视图可以保存在客户端(通常存储在隐藏域中)或服务器中(通常在会话对象中)。

2.4　创建一个简单的 JSF 应用

通过一个具有登录功能的 JSF 应用，熟悉创建 JSF 应用的一般过程，了解 JSF 应用的组成。

2.4.1　登录应用

这是一个实现登录功能的 JSF 应用，项目名称为 ch2_login。当应用运行时，首先显示一个登录页面，要求用户输入用户名和密码。用户提交用户名和密码后，应用验证用户名和密码是否正确。如果用户名和密码正确，应用显示一个登录成功页面，否则显示一个登录失败页面。应用的运行效果如图 2-4～图 2-6 所示。



图 2-4　login 的登录页面



图 2-5　login 的登录成功页面



图 2-6 login 的登录失败页面

创建本应用的具体步骤如下(在“项目”窗口中进行)。

(1) 从“文件”菜单,选择“新建项目”命令,打开“新建项目”对话框。

(2) 选择项目类型：Java Web|Web 应用程序。

(3) 指定项目名称：ch2_login,然后根据需要选择存放项目的位置。勾选“设置为主项目”复选框。

(4) 选择服务器：GlassFish Server 3,选择 Java EE 版本：Java EE 6 Web。

(5) 选择要使用的框架：JavaServer Faces。在 JavaServer Faces 配置中,首选页面语言采用默认的 Facelets。

(6) 单击“完成”按钮,完成应用项目的新建。

2.4.2 创建模型

按照 MVC 架构,模型实现应用的业务逻辑。实现业务逻辑的软件组件可以是普通的 Java 类,也可以 JSF 中的受管 bean 或 Java EE 中的 EJB。本书主要采用一些简单的 Java 类来实现所需的业务逻辑。

该应用的业务逻辑是根据用户名和密码,验证用户身份是否合法,或者根据用户名,返回用户相应的密码。为此,创建了两个 Java 类：DataBase.java(代码清单 2-1)模拟数据库,存放注册用户的用户名和密码信息;UserManager.java(代码清单 2-2)提供唯一的一个业务方法 getPassword(String),该方法根据指定的用户名返回相应的密码。如果用户名不合法,方法返回 null。两个类都存放在 Java 包 model 中。

代码清单 2-1 model.DataBase

```
package model;
import java.util.HashMap;
import java.util.Map;

public class DataBase {
  private static final Map<String,String>users =new HashMap<String,String>();
  static {
    users.put("李晓彤","121212");
    users.put("刘金辉","123456");
    users.put("James","666666");
  }
  public static Map<String,String>getUsers(){
    return users;
```

```
    }
}
```

代码清单 2-2　model.UserManager

```
package model;
import java.util.Map;

public class UserManager {
  public String getPassword(String username){
    Map<String,String>users =DataBase.getUsers();
    return users.get(username);
  }
}
```

创建本应用模型的具体步骤如下(在“项目”窗口中进行)。

(1) 创建 Java 包。

① 单击选择项目的“源包”结点。

② 从“文件”菜单,选择“新建文件”命令,打开“新建文件”对话框。

③ 选择文件类型:Java|Java 包。

④ 指定包名:model,然后单击“完成”按钮。

(2) 创建 Java 类。

① 单击选择存放 Java 类的包结点 model。

② 从“文件”菜单,选择“新建文件”命令,打开“新建文件”对话框。

③ 选择文件类型:Java|Java 类。

④ 指定类名:DataBase,然后单击“完成”按钮。

⑤ 根据代码清单 2-1,在编辑器窗格中完成对该 Java 类的定义。

用同样的方法可以创建 UserManager.java 文件。

2.4.3　创建支撑 bean

支撑 bean 是一种受管 bean,它在 JSF 页面和模型之间起着沟通的桥梁作用。有关受管 bean 的详细内容将在第 3 章做进一步介绍。

本应用中,定义了一个文件名为 Login.java 的支撑 bean(代码清单 2-3),其作用有以下几个方面。

(1) 通过 name 属性和 pw 属性的 setter 方法接收并保存用户输入的用户名和密码。

(2) 动作方法 action 通过调用业务方法判断当前用户的身份是否合法。若用户名或密码有错,设置相应的错误信息(errmsg)。方法根据判断结果返回相应的字符串。

该动作方法会在用户提交登录请求时自动调用,方法的返回的结果会作为导航处理器进行导航的依据。

(3) 登录成功页面会通过 name 属性的 getter 方法获取用户名并显示。登录失败页面会通过 errmsg 属性的 getter 方法获取错误提示信息并显示。

代码清单 2-3　支撑 bean（Login. java）

```
package bean;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import model.UserManager;

@ManagedBean
@RequestScoped
public class Login {

  private String name;
  public String getName(){
    return name;
  }
  public void setName(String name){
    this.name=name;
  }

  private String pw;
  public String getPw(){
    return pw;
  }
  public void setPw(String pw){
    this.pw=pw;
  }

  private String errmsg;
  public String getErrmsg(){
    return errmsg;
  }

  public String action(){
    UserManager um=new UserManager();
    String password=um.getPassword(name);
    if(password==null){
      errmsg="用户名不存在。";
      return "failure";
    } else if(!password.equals(pw)){
      errmsg="密码错误。";
      return "failure";
    }
    return "success";
  }
}
```

上述类定义中，标注@ManagedBean 指明这是一个受管 bean 类，标注@RequestScoped 指明该 bean 的范围是请求。默认情况下，bean 的名称与类名是一致的（第 1 个字母为小写），如类名为 Login，bean 的默认名称为 login。如果 bean 的名称与类名不一致，可在标注@ManagedBean 中用 name 属性指定，如@ManagedBean(name="me")。

创建本应用支撑 bean 的具体步骤如下（在“项目”窗口中进行）。

（1）创建 Java 包。

① 单击选择项目的“源包”结点。

② 从“文件”菜单，选择“新建文件”命令，打开“新建文件”对话框。

③ 选择文件类型：Java|Java 包。

④ 指定包名：bean，然后单击“完成”按钮。

（2）创建支撑 bean。

① 单击选择存放支撑 bean 的包结点 bean。

② 从“文件”菜单，选择“新建文件”命令，打开“新建文件”对话框。

③ 选择文件类型：JavaServer Faces|JSF 受管 Bean。

④ 指定类名：Login，bean 名称取默认的 login，范围取默认的 request，然后单击“完成”按钮。

⑤ 根据代码清单 2-3，在编辑器窗格中完成对该 bean 类的定义。

2.4.4 创建 JSF 页

采用 Facelets 技术的 JSF 页面是一种 XHTML 文件。login 应用一共有 3 个页面。

- 登录页面：login. xhtml；
- 登录成功页面：success. xhtml；
- 登录失败页面：failure. xhtml。

login. xhtml 页（代码清单 2-4）主要包含一个用户名文本域、一个密码文本域和一个提交按钮。其中 #{login. name} 和 #{login. pw} 是两个 EL 表达式。JSF 页通过 EL 表达式与支撑 bean 建立关联。比如 EL 表达式 #{login. name}，login 是 bean 的名称，name 是 bean 的一个属性。当用户提交登录时，输入的用户名将被保存到 login 支撑 bean 的 name 属性中。

代码清单 2-4　登录页面（login. xhtml）

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>ch2_login</title>
  </h:head>
  <h:body>
    <h:form>
      <h:outputLabel for="username" value="用户名"/>
```

```
        <h:inputText id="username" value="#{login.name}" maxlength="15"/>
        <p></p>
        <h:outputLabel for="password" value="密 码"/>
        <h:inputSecret id="password" value="#{login.pw}" maxlength="15"/>
        <p></p>
        <h:commandButton id="reset" value="重置" type="reset"/>
        <h:commandButton id="submit" value="登录" action="#{login.action}"/>
      </h:form>
    </h:body>
</html>
```

success. xhtml 页(代码清单 2-5)主要显示一个欢迎信息,其中 EL 表达式#{login. name}会从 login 支撑 bean 中获取 name 属性的值并显示。

代码清单 2-5　登录成功页面(success. xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
      <title>ch2_login</title>
    </h:head>
  <h:body>
    你好:#{login.name},欢迎登录。
  </h:body>
</html>
```

登录失败页面 failure. xhtml(代码清单 2-6)主要显示一个登录失败的信息,其中 EL 表达式#{login. errmsg}会从 login 支撑 bean 中获取 errmsg 属性的值并显示。

代码清单 2-6　登录失败页面(failure. xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
      <title>ch2_login</title>
    </h:head>
  <h:body>
    登录失败!#{login.errmsg}
  </h:body>
</html>
```

创建 JSF 页面的一般步骤如下(在“项目”窗口中进行)。

(1) 单击选择项目结点。
(2) 从“文件”菜单,选择“新建文件”命令,打开“新建文件”对话框。
(3) 选择文件类型:JavaServer Faces|JSF 页。
(4) 指定 JSF 页的文件名(不需要指定扩展名),有选择地指定文件夹。
(5) 单击“完成”按钮。

2.4.5 设置上下文路径

如果 JSF 应用以 GlassFish 为部署服务器,则其上下文路径可在自动产生的 sun-web.xml 文件中用 context-root 元素指定。

```
<context-root>/login</context-root>
```

默认情况下,JSF 应用的上下文路径为斜杠/跟应用项目的名称。在本例中,项目名称为 ch2_login,应用的默认上下文路径为/ch2_login。在此,把上下文路径改为/login。

2.4.6 检查部署描述符

在 NetBeans IDE 环境下创建一个 JSF 应用时,会自动产生部署描述符 web.xml 文件。该文件位于 WEB-INF 下,包含着若干对 JSF 应用来说必要或有用的元素。

1. 上下文参数指定项目阶段

```
<context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
</context-param>
```

该参数指定应用项目在其生命周期中所处的阶段,有效的取值包括 Development、UnitTest、SystemTest 和 Production。在开发阶段(Development),JSF 框架会在应用运行出错时给用户提供更多的调试信息。

2. servlet 元素和相应的 servlet-mapping 元素指定 FacesServlet

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

上述元素指定了 FacesServlet 的名称和 url 模式。默认情况下,FacesServlet 的 url 模式是/faces/*。FacesServlet 是 JSF 框架入口点,所有对 JSF 页面的请求都必须通过它。所以如果访问 JSF 页 login.xhtml,不能简单地指定 URL 为 http://localhost:8080/login/login.xhtml,而应指定为 http://localhost:8080/login/faces/login.xhtml。

3. welcome-file-list 元素指定欢迎页面

```
<welcome-file-list>
    <welcome-file>faces/login.xhtml</welcome-file>
</welcome-file-list>
```

欢迎页面又称主页或默认页面，通常是指用户访问一个网站或 Web 应用(没有指定具体页面)时，被第一个显示的页面。

默认情况下，在 NetBeans IDE 下创建一个 JSF 应用时，会自动创建一个 index. xhtml 页面文件，且指定该文件为欢迎页面。这里，将欢迎页面改为登录页面 login. xhtml，同时可以将 index. xhtml 文件删除。

要删除 index. xhtml 文件，可以在"项目"窗口中右击该文件结点，然后在打开的快捷菜单中选择"删除"命令。

说明：通常，如果没有指定欢迎页面，Web 服务器会自动把保存在 Web 应用文档根目录下的 index. html 文件作为默认页面。

2.4.7 运行 JSF 应用

如果登录应用是主项目(项目结点粗体显示)，可以直接按 F6 键或单击工具栏上的"运行主项目"按钮运行该项目。如果 hello 应用不是主项目，可以在"项目"窗口中右击该项目结点，然后在打开的快捷菜单中选择"运行"命令运行该项目。

当运行一个 JSF 应用项目时，NetBeans IDE 将完成以下工作。

(1) 对项目中文件的任何修改，自动进行保存。

(2) 将 JSF 应用重新部署到服务器。如果服务器还没有启动，则先启动服务器。

(3) 通过浏览器向 JSF 应用发出请求(http://localhost:8080/login/)。如果浏览器还没有打开，则先打开浏览器。

登录应用项目运行后，首先显示欢迎页面，即登录页面。当用户提交登录请求时，如果用户名和密码都合法，将显示登录成功页面，否则显示登录失败页面。

2.5 小　　结

- JSF 是一种基于 Java 的 Web 应用的用户界面软件框架，提供了一种以组件为中心、事件驱动的用户界面构建方法。
- MVC 架构将软件分割为松耦合的模型、视图、控制器 3 个部分，各部分各司其职。基于 JSF 框架，开发人员可以很好地遵循 MVC 设计架构开发 Web 应用。
- JSF 页面由标记组成，每个标记在服务器端的内部表示成一个组件实例。一个 JSF 页面的所有标记的组件实例组成一棵组件树，是 JSF 页面在服务器端的内部表示。
- 呈现器既负责从服务器端组件到客户端响应的转换，也负责对请求参数进行解析并保存到服务器端特定的组件内。
- JSF 请求处理生命周期包括恢复视图、应用请求值、处理验证、更新模型值、调用应用和呈现响应这 6 个阶段。

习　题　2

1. 什么是 JSF?

2. 什么是 MVC 设计架构? MVC 设计架构有何优点?

3. 术语解释:

- JSF 标记、JSF 组件;
- JSF 页面、JSF 视图;
- 呈现器;
- JSF 组件标识符、JSF 组件客户端标识符。

4. 简述 JSF 请求处理生命周期。

5. 创建一个 JSF 应用(sh2_palindrome),包含 index. xhtml 和 result. xhtml 两个 JSF 页面。index. xhtml 包含一个表单,用户可以输入并提交一个字符串,如图 2-7 所示。



图 2-7　第 5 题示意图

当用户提交字符串后,JSF 应用将判断该字符串是否为“回文”,然后转 result. xhtml 页显示判断结果。result. xhtml 页显示的内容包括用户输入的字符串以及判断结果:“是回文”或“不是回文”。

第 3 章　受管 bean 与 EL 表达式

本章主题：

- 编写 bean 类
- 配置受管 bean
- 值表达式
- 方法表达式
- 在页面外使用 EL 表达式

所谓受管 bean 是指受 JSF 框架的受管 bean 工具管理的 JavaBean。这里，开发人员除需按一定规范编写 bean 类外，还需对受管 bean 进行相关的配置声明，而 bean 实例的创建和清除则由受管 bean 工具根据需要自动完成。

按作用分，受管 bean 可大致分为支撑 bean（backing bean）和模型 bean 两大类。支撑 bean 与 JSF 页面相关联，其属性往往与页面的组件或组件属性绑定在一起。模型 bean 实现应用的业务逻辑。在 Java EE 架构中，业务逻辑通常由 EJB 实现，但在中小型的 JSF 应用中，业务逻辑完全可以由受管 bean 或者普通的 Java 类实现。本书介绍的应用示例中，主要用受管 bean 作为页面的支撑 bean，用普通的 Java 类实现应用所需的业务逻辑。

EL（Expression Language）技术源于 JSP 中的表达式语言，在 JSF 中得到了进一步的扩展。EL 表达式通常应用于页面，通过 EL 表达式，页面可以方便地访问受管 bean 属性或调用受管 bean 方法。在 JSF 中，EL 表达式具有举足轻重的作用，它与支撑 bean 一起共同构筑成表示层与业务层之间的通信桥梁。

3.1　编写 bean 类

与普通的 JavaBean 类一样，受管 bean 类的编写也需要符合一定的规范。下面是编写 bean 类的一个基本约定：

（1）有一个不带形参的 public 构造方法。

（2）提供用于获取和/或设置属性值的 public 方法。这些方法的名称形如 getXxx、setXxx，其中 Xxx 为相应的属性名，getter 方法的返回类型为相应属性的类型。如果属性类型为 boolean，那么方法名 getXxx 也可用 isXxx 代替。

（3）bean 类一般应该实现 java.io.Serializable 接口，即 bean 实例应该是可序列化的。尤其是作用域为应用、会话和视图的受管 bean，将其声明为可序列化的，可便于应用服务器对其进行有效管理。

与普通的 Java 类实例由 new 表达式创建不同，bean 实例通常由受管 bean 工具自动创建。受管 bean 工具一般调用不带形参的构造方法创建 bean 实例，因此 bean 类必须提供该构造方法。

属性是 bean 的一个基本特性。工具通过 bean 类提供的 getter 方法和 setter 方法，了

解 bean 所具有的属性，以及实现对相应属性的访问。一个提供 getter 方法的属性被认为是可读的，一个提供 setter 方法的属性被认为是可写的。

在很多情况下，一个属性会有一个对应的实例变量，但并非一定如此。请看下面的 bean 类（代码清单 3-1）。

代码清单 3-1　受管 bean 类示例（Circle.java）

```
package bean;
import java.io.Serializable;

public class Circle implements Serializable {
  private int radius;
  public void setRadius(int r){
    radius=r;
  }
  public int getRadius(){
    return radius;
  }

  public double getArea(){
    return Math.PI * radius * radius;
  }
}
```

该 bean 类定义了一个名为 radius 的可读写属性，以及一个名为 area 的只读属性。这里 area 属性并没有对应的实例变量。

当然，bean 类也是一种 Java 类，bean 实例也是一种 Java 对象。可以在 bean 类中声明一些普通的实例方法，并通过 bean 实例调用这些方法。

3.2　配置受管 bean

JSF 受管 bean 工具负责管理受管 bean，包括受管 bean 的创建、初始化、清除，以及客户代码对受管 bean 的访问。JSF 受管 bean 工具根据配置信息管理受管 bean，开发人员负责声明受管 bean 的配置。

3.2.1　声明受管 bean

受管 bean 的配置信息如下。

（1）bean 名称：指 bean 实例的名称，外界通过该名称访问指定 bean 的属性或调用指定 bean 的方法。在一个 JSF 应用中，每个受管 bean 的名称应该是唯一的。

（2）bean 类名：由包名限制的完整类名。当外界通过 bean 名称访问 bean 时，如果 bean 不存在，系统将自动调用该 bean 类的不带形参的构造方法创建一个 bean 并进行初始化，然后保存在指定的作用域内。如果 bean 已经存在，就直接使用它。

（3）作用域：指 bean 的生存期限。受管 bean 的生存期限包括：应用、会话、视图、请

求、无。

(4) 属性初值：JSF 受管 bean 工具能够据此信息对受管 bean 进行初始化。初始化发生于 bean 实例创建后和投入使用之前。

受管 bean 的配置信息有两种声明方式：一是在 bean 类中用适当的 Java 标注(annotation)声明；二是在 JSF Faces 配置文件(XML 文件)中用相应的元素声明。

下面是用 Java 标注声明受管 bean 的例子。

```
package bean;
import java.io.Serializable;

@ManagedBean(name="me")
@RequestScoped
public class UserBean implements Serializable {
…
}
```

标注@ManagedBean 指明它所标注的 UserBean 类是一个 bean 类，其中 name 属性指定 bean 的名称为 me。默认情况下，bean 的名称与类名一致，且第一个字母是小写。也就是说，在该例中如果@ManagedBean 标注没有设置 name 属性，则 bean 的名称将为 userBean。

标注@RequestScoped 指明受管 bean 的作用域为请求。各作用域对应的 Java 标注如下：

- 应用：@ApplicationScoped；
- 会话：@SessionScoped；
- 视图：@ViewScoped；
- 请求：@RequestScoped；
- 无：@NoneScoped。

下面是在 JSF Faces 配置文件中用相应元素声明受管 bean 的例子。

```
<faces-config version="2.0"......>
    <managed-bean>
        <managed-bean-name>me</managed-bean-name>
        <managed-bean-class>bean.UserBean</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>
    …
</faces-config>
```

managed-bean 是顶层 faces-config 元素的子元素。每个 managed-bean 元素声明一个受管 bean。其中，managed-bean-name 子元素指定 bean 名称，managed-bean-class 子元素指定 bean 类名，managed-bean-scope 子元素指定 bean 的作用域。各作用域对应的取值如下：

- 应用：application；

- 会话：session；
- 视图：view；
- 请求：request；
- 无：none。

上述元素都应该放置在 JSF Faces 配置文件中。在 NetBeans IDE 中，创建一个配置文件的步骤如下：

① 单击选择项目结点。

② 从“文件”菜单，选择“新建文件”命令，打开“新建文件”对话框。

③ 选择文件类型：JavaServer Faces|JSF Faces 配置。

④ 指定文件名及存放位置，然后单击“完成”按钮。

一个 JSF 应用项目可以包含多个配置文件，但最常用的是保存在 WEB-INF 目录下的名为 faces-config.xml 的配置文件。

3.2.2 受管 bean 的作用域

前面提到受管 bean 的作用域包括应用、会话、视图、请求和无。这里对各种作用域的含义及特点进行介绍。

1. 应用作用域

该作用域的受管 bean 在 Web 应用的整个存活期间可用，被该应用的所有客户、会话和请求共享。

Web 应用的存活期始于应用被部署到应用服务器，止于应用被取消部署。应用作用域的受管 bean 实例通常在第一次被访问时创建，并一直保持活动，直到 Web 应用被取消部署。期间，不同的客户、会话和请求通过 EL 表达式或其他方式访问该受管 bean 时，访问到的都是同一个 bean 实例。

一个 Web 应用可以被多个客户并发访问，也可以包含多个会话。当需要在多个客户或多个会话之间共享信息或传递信息时，可以考虑使用应用作用域的受管 bean。

2. 会话作用域

该作用域的受管 bean 在会话的存活期间可用，可被该会话期间的所有请求共享。

会话是指一个客户对一个 Web 应用的一系列请求。这一系列的请求并不要求是连续的，期间可以访问其他的 Web 应用或网站。在 JSF 中，会话通常始于客户对一个 JSF 应用的第一次访问，止于会话被显式结束（调用 HttpSession 对象的 invalidate 方法）、会话超时或者浏览器关闭。会话作用域受管 bean 实例通常在会话期间该受管 bean 第一次被访问时创建，并在会话期间一直保持活动。在一个会话期间各请求通过 EL 表达式访问一个会话作用域的受管 bean 时，访问到的都是同一个 bean 实例。会话结束时，在该会话期间创建的所有会话作用域受管 bean 实例将被销毁。

因为在 JSF 应用运行期间，可以有多个会话同时存在，所以同一个会话作用域的受管 bean 类就可能有多个 bean 实例同时同在。不同的会话访问不同的 bean 实例。

当需要在一个会话的各请求之间共享或传递信息时，可以考虑使用会话作用域的受管 bean。例如，一次网上购物过程通常是一个会话，每次请求会将一些商品放入购物车（通常是 List 对象或 Map 对象），最后再进行结账。这里可以将购物车保存在会话作用域的受管

bean 中。

3. 视图作用域

该作用域的受管 bean 仅在当前视图内可用。在 JSF 中,JSF 页面又称为视图。视图作用域始于对一个新的 JSF 页面的请求处理、或者导航到一个新的 JSF 页面,止于对另一个 JSF 页面的请求处理、或者导航到另一个 JSF 页面。视图作用域受管 bean 实例在视图作用域期间该受管 bean 第一次被访问时创建,并在该作用域期间一直保持活动。在视图作用域期间发生的对某视图作用域的受管 bean 的访问,访问到的都是同一个 bean 实例。视图作用域结束时,该作用域期间创建的所有视图作用域受管 bean 实例将被销毁。

对典型的回送请求,可以把请求处理的 6 个阶段分为两个过程。前 5 个阶段为第一过程,处理请求参数、调用业务逻辑;第 6 个阶段为第二个过程,完成呈现响应。第一个过程涉及的视图称为源视图,通常是上一次请求处理产生的响应视图。此时,上次请求处理的第二个过程和本次请求处理的第一个过程就处于同一个视图作用域。第二个过程涉及的视图称为目标视图,与源视图可能相同也可能不相同,取决于导航结果。如果目标视图与源视图相同,则当前视图作用域就从第一个过程延伸到第二个过程,否则,当前视图作用域结束,新的视图作用域开始。

4. 请求作用域

该作用域的受管 bean 在请求处理期间有效,它始于 JSF 框架接收一个客户请求并着手处理,终于处理产生的响应返回客户端。

请求作用域受管 bean 实例在请求作用域期间该受管 bean 第一次被访问时创建,并在该作用域期间一直保持活动,可被请求处理过程中涉及的源视图和目标视图共享。请求作用域结束时,该作用域期间创建的所有请求作用域受管 bean 实例将被销毁。

5. 无(none)作用域

作用域声明为 none 的受管 bean 没有特定的作用域。当通过 EL 表达式访问此种受管 bean 时,JSF 受管 bean 工具负责创建 bean 实例并返回,但不会进一步对 bean 实例进行维护,即不负责 bean 实例的保存以及销毁。

JSF 受管 bean 工具本身并不支持线程安全机制。由于视图作用域、请求作用域和无作用域的受管 bean 通常是单线程的,所以能够确保线程安全。但应用作用域和会话作用域的受管 bean 却是可以多线程的,例如在一个会话期间,客户可以同时从多个浏览器窗口提交请求,并访问同一个会话作用域受管 bean。如果要在应用作用域和会话作用域的受管 bean 中保证线程安全,应该提供锁机制。

3.2.3 视图作用域受管 bean 应用示例

该应用项目(ch3_viewscope)主要由两个页面和一个受管 bean 组成。项目的运行效果如图 3-1 所示。

项目运行时,欢迎页面上首先显示一个单词集,以及一个文本框和一个提交按钮。用户可以在文本框中输入单词集中任意一个单词并提交。如果用户输入的单词正确,该单词会从单词集中删除。当单词集中所有单词都删除后,项目显示包含一个“再来一次”命令按钮的页面。

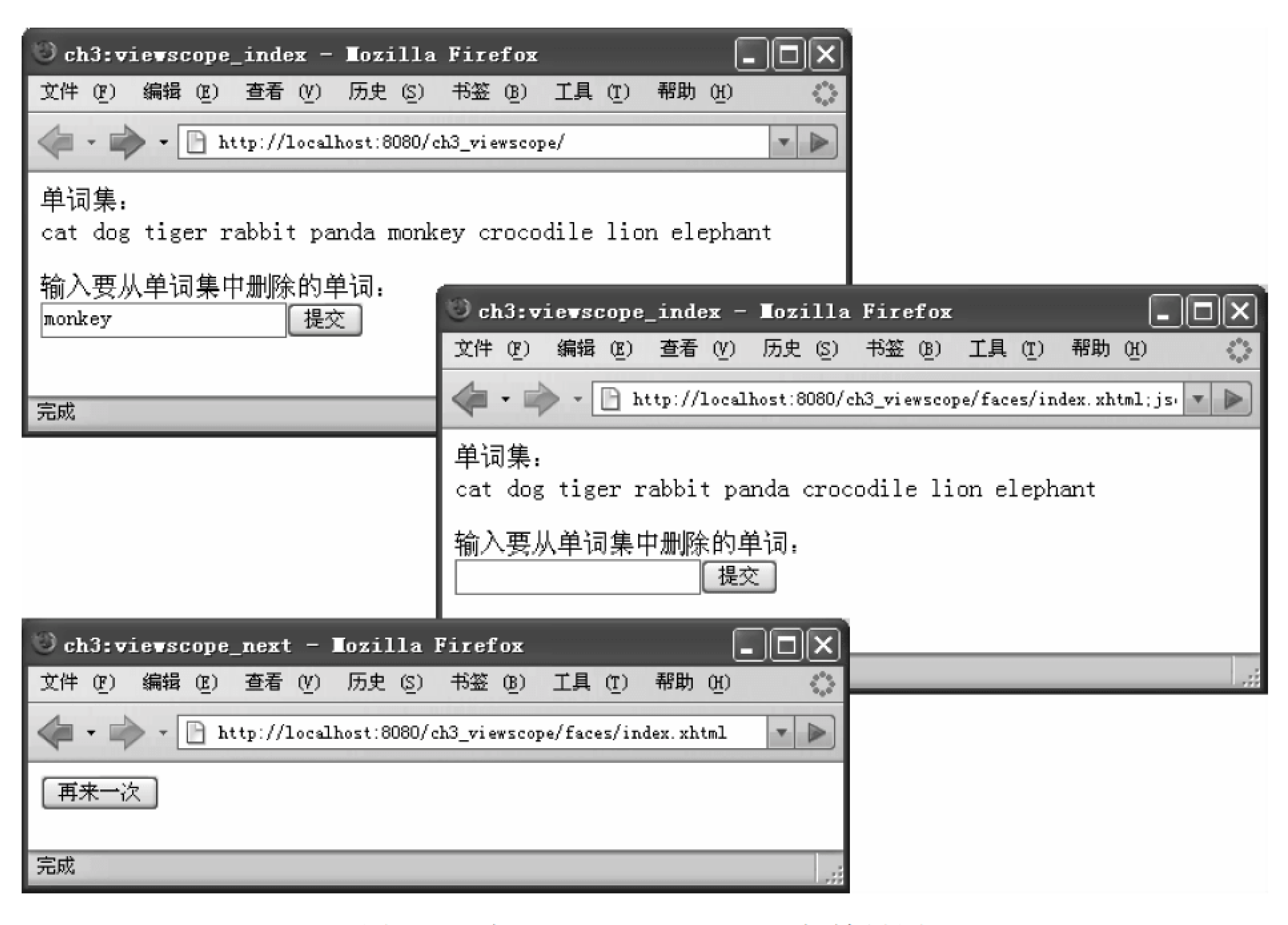


图 3-1　应用 ch3_viewscope 运行效果图

1. 受管 bean

本应用仅定义了一个受管 bean 类，文件名为 Index.java，见代码清单 3-2。一方面，该受管 bean 起着支撑 bean 的作用。其中 words 属性保存着页面要显示的单词集。对于页面来说，这是一个只读属性。parameter 属性可以接收和保存用户输入的单词。action 方法在用户单击"提交"按钮时被调用，方法的返回的结果会作为导航处理器进行导航的依据。

另一方面，该受管 bean 也扮演了模型 bean 的作用。该应用涉及的业务处理较为简单，只是从一个单词集中去除一个单词，因此没有定义单独的模型类或模型受管 bean。有关的数据处理代码直接放置在了该受管 bean 的 action 方法中。

代码清单 3-2　受管 bean(Index.java)

```java
package bean;

import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;

@ManagedBean
@ViewScoped
public class Index implements Serializable {

  public Index(){
    words="cat dog tiger rabbit panda monkey crocodile lion elephant ";
  }

```

```
    private String words;
    public String getWords(){
      return words;
    }

    private String parameter;
    public void setParameter(String in){
      parameter=in.trim()+" ";
    }
    public String getParameter(){
      return "";
    }

    public String action(){
      int n=parameter.length();
      int l=words.length();
      int s=words.indexOf(parameter);
      if(s!=-1){
        if(s+n>=l){
          words=words.substring(0,s);
        } else {
          words=words.substring(0,s)+words.substring(s+n);
        }
      }
      if(words.length()==0){
        return "next";
      }
      return null;
    }
}
```

标注@ViewScoped 指明这是一个视图作用域的受管 bean。bean 实例在欢迎页面第一次呈现响应时创建。只要用户还没有从单词集中去除所有的单词,应用不会导航到第二个页面。在这期间,该 bean 实例一直存在。

2. JSF 页面

该应用包含两个页面。index. xhtml 页(代码清单 3-3)是一个欢迎页面。该页面通过 EL 表达式 #{index. words}显示受管 bean 的 words 属性,通过 EL 表达式 #{index. parameter}将用户的输入保存到受管 bean 的 parameter 属性中,通过 EL 表达式 #{index. action}在用户提交时调用受管 bean 的 action 方法。

代码清单 3-3　index. xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
```

```
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>ch3:viewscope_index</title>
  </h:head>
  <h:body>
    <h:form>
      单词集：<br/>
      <h:outputText value="#{index.words}"/>
      <p/>
      输入要从单词集中删除的单词：<br/>
      <h:inputText value="#{index.parameter}"/>
      <h:commandButton value=" 提交 " action="#{index.action}"/>
    </h:form>
  </h:body>
</html>
```

next.xhtml 页(代码清单 3-4)仅包含一个命令按钮，单击该按钮将重新回到 index.xhtml 页面。

代码清单 3-4　next.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>ch3:viewscope_next</title>
  </h:head>
  <h:body>
    <h:form>
      <h:commandButton value=" 再来一次 " action="index"/>
    </h:form>
  </h:body>
</html>
```

当用户从单词集中去除掉所有的单词后，应用将从欢迎页面导航到该页面，此时上述 bean 实例被销毁。当用户从该页面回到欢迎页面时，新的 bean 实例将被创建。

3.2.4　生命周期方法

在定义受管 bean 类时，可以通过相应的 Java 标注指定受管 bean 的一些生命周期方法：

```
public class MyBean implements Serializable {
    @PostConstruct
    public void initialize(){
        //初始化代码
    }
```

```
    @PreDestroy
    public void shutdown(){
        //关闭代码
    }
}
```

标注@PostConstruct 用于修饰一个方法，该方法会在 bean 实例创建之后被自动调用。标注@PreDestroy 用于修饰一个方法，该方法会在 bean 实例销毁之前被自动调用。

由于 JSF 受管 bean 工具并不负责无(none)作用域受管 bean 实例的保存和销毁，所以对这种类型的受管 bean，标注@PreDestroy 没有意义。

3.2.5 初始化受管 bean

JSF 受管 bean 工具会在需要时自动创建 bean 实例，并在将 bean 实例投入使用之前根据配置信息初始化 bean 实例。

1. 简单的属性设置

下面是一个简单的属性设置示例。其中 managed-property 是 managed-bean 元素的可选子元素，用于为一个 bean 的一个元素设置初始值。property-name 是 managed-property 元素的必选子元素，指定需要设置的属性的名称。value 和 null-value 都是 managed-property 元素的子元素，前者可以为属性指定一个简单值，后者为属性指定 null 值。

```
<managed-bean>
    …
    <managed-property>
        <property-name>name</property-name>
        <value>louby</value>
    </managed-property>
    <managed-property>
        <property-name>pw</property-name>
        <null-value/>
    </managed-property>
</managed-bean>
```

受管 bean 工具会根据属性本身推定属性的类型。如果属性类型不是 String，那么工具会首先将值转换成所需的类型，然后再设置属性。

2. 初始化 List 型属性

如果属性类型为 List，那么可以使用 list-entries 元素为属性设置初值。这里 list-entries 元素是 managed-property 元素的一个子元素。

list-entries 元素至多有一个 value-class 子元素，用于指定要添加到 List 表中的元素的数据类型。list-entries 元素可以有多个 value 子元素，每个 value 子元素指定 List 表的一个元素。下面例子对一个 List 型的 names 属性设置初值。

```
<managed-bean>
    …
    <managed-property>
```

```
            <property-name>names</property-name>
            <list-entries>
                <value-class>java.lang.String</value-class>
                <value>liu</value>
                <value>zhao</value>
                <value>wang</value>
            </list-entries>
        </managed-property >
    </managed-bean>
```

说明：

(1) 如果 bean 实例化后该属性已经指向一个表，那么工具只是将所列元素值按序添加到表原有值的后面。如果 bean 实例化后该属性没有值(即为 null 值)，工具将创建一个 ArrayList 表，并将所列元素值按序添加到表中，然后设置该属性。

(2) 默认情况下，工具添加到表中的元素都是 java. lang. String 型的，若为其他类型，应该用 value-class 元素指定所需类型。

3. 初始化 Map 型属性

如果属性类型为 Map，那么可以使用 map-entries 元素为属性设置初值。这里 map-entries 是 managed-property 元素的一个子元素。

map-entries 元素至多有一个 key-class 子元素和一个 value-class 子元素，但可以有多个 map-entry 子元素。其中，key-class 子元素指定键的数据类型，value-class 子元素指定值的数据类型。每个 map-entry 子元素指定一个键—值对，其 key 子元素指定键，value 子元素指定值。下面例子为一个 Map 型的 cart 属性设置初值。

```
    <managed-bean>
        …
        <managed-property>
            <property-name>cart</property-name>
            <map-entries>
                <key-class>java.lang.String</key-class>
                <value-class>java.lang.Integer</value-class>
                <map-entry>
                    <key>101</key>
                    <value>1</value>
                </map-entry>
                <map-entry>
                    <key>102</key>
                    <value>2</value>
                </map-entry>
            </map-entries>
        </managed-property >
    </managed-bean>
```

说明：

(1) 如果 bean 实例化后该属性已经指向一个映射，那么工具只是将所列键—值对添加

到映射中。如果 bean 实例化后该属性没有值(即为 null 值),工具将创建一个 HashMap 映射,并添加所列的键—值对,然后设置该属性。

(2) 默认情况下,工具产生的键和值都是 java. lang. String 型的,若为其他类型,应该用 key-class 和 value-class 元素指定所需的类型。

3.2.6 List 和 Map 型受管 bean

表(List)或映射(Map)既可以作为受管 bean 的属性并通过配置信息进行初始化,也可以被 managed-bean 元素声明为受管 bean 本身并进行初始化。

当用 managed-bean 元素声明一个 List 或 Map 型的受管 bean 时,managed-bean-class 子元素必须指定实现 List 接口或 Map 接口的一个具体类,如 ArrayList、HashMap 等。list-entries 和 map-entries 子元素可以分别初始化该 List 或 Map 型的受管 bean。这里,list-entries 和 map-entries 是 managed-bean 元素的直接子元素。

下面例子声明了一个 List 型受管 bean,具体的类型是 ArrayList,初始值包含 3 个元素。在 EL 表达式中,可以通过 bean 名称 names 访问该 List 型受管 bean。

```
<managed-bean>
    <managed-bean-name>names</managed-bean-name>
    <managed-bean-class>java.util.ArrayList</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <list-entries>
        <value>liu</value>
        <value>zhao</value>
        <value>wang</value>
    </list-entries>
</managed-bean>
```

默认情况下,表的元素类型、映射的键和值的类型都是 String。若需其他类型,应该在 list-entries 或 map-entries 元素中用 value-class 和 key-class 子元素进行指定。

3.2.7 初始化受管 bean 应用示例

该应用项目(ch3_initializebean)主要由一个页面和一个受管 bean 组成。项目的运行效果如图 3-2 所示。



图 3-2 应用 ch3_initializebean 运行效果图

受管 bean 包含一个 List 型属性，存放着一组姓名。页面根据索引值显示相应的姓名，其中索引值可以由用户在文本域中指定。

1. 创建配置文件

默认情况下，新创建的 JSF 应用项目不包含 JSF 配置文件。在 NetBeans IDE 中，可以按下列步骤创建一个配置文件。

(1) 单击选择项目结点。

(2) 从“文件”菜单，选择“新建文件”命令，打开“新建文件”对话框。

(3) 选择文件类型：JavaServer Faces|JSF Faces 配置，单击“下一步”按钮。

(4) 如果项目还没有包含任何 Faces 配置文件，那么通常应该默认 NetBeans 提供的文件名和存放位置，即文件名为 faces-config，存放位置为 WEB-INF 目录。

(5) 最后单击“完成”按钮。

本应用中，首先按上述步骤创建配置文件 WEB-INF\faces-config.xml。

2. 创建受管 bean

本应用仅包含一个受管 bean(代码清单 3-5)，其配置信息不采用 Java 标注形式声明，而是放置在配置文件中。

用 NetBeans IDE 中的命令新建一个受管 bean 时，若在“名称和位置”对话框中选择“向配置文件添加数据”复选框，那么有关该受管 bean 的类名、bean 名称和作用域等配置信息就不会以 Java 标注形式出现在 bean 类中，而是出现在指定的配置文件中。

这里，将受管 bean 的类名设置为 bean.NameList(其中 bean 是包名)，bean 名称设置为 names，bean 的作用域设置为 request。

代码清单 3-5　NameList.java

```
package bean;
import java.util.List;

public class NameList {

  public NameList(){
  }

 private int index;
  public int getIndex(){
    return index;
  }
  public void setIndex(int index){
    this.index=index;
  }

  private List<String>list;
  public void setList(List<String>list){
    this.list=list;
  }
```

```

    private String element;
    public String getElement(){
      return list.get(index);
    }
}
```

该受管 bean 包含一个可读写的 index 属性，一个可写的 list 属性，以及一个可读的 element 属性。

3. 配置受管 bean

配置文件 faces-config.xml 中有关受管 bean 的声明和配置信息如代码清单 3-6 所示。其中，受管 bean 的类名、bean 名称及作用域等信息在创建受管 bean 时自动添加，而受管 bean 的初始化信息由手工编制完成。

这里，names 受管 bean 的 index 属性值初始化为 0，list 属性初始化后包含 4 个元素(姓名)。

代码清单 3-6　faces-config.xml

```
<managed-bean>
  <managed-bean-name>names</managed-bean-name>
  <managed-bean-class>bean.NameList</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>index</property-name>
    <value>0</value>
  </managed-property>
  <managed-property>
    <property-name>list</property-name>
    <list-entries>
      <value>李小明</value>
      <value>刘宇翔</value>
      <value>胡刚</value>
      <value>赵日杰</value>
    </list-entries>
  </managed-property>
</managed-bean>
```

4. JSF 页面

本应用只有一个页面(代码清单 3-7)，其中 inputText 组件标记的 value 属性与 bean 的可读写属性 index 绑定在一起，outputText 组件标记的 value 属性与 bean 的可读属性 element 绑定在一起。

代码清单 3-7　index.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html">
 <h:head>
   <title>initializebean</title>
 </h:head>
 <h:body>
   <h:form>
     <h:outputLabel id="l1" for="i1" value="输入索引值 "/>
     <h:inputText id="i1" value="#{names.index}"/>
     <p></p>
     <h:commandButton id="c1" value="确定"/>
     <p></p>
     <h:outputText id="o1" value="相应的姓名 #{names.element}"/>
   </h:form>
 </h:body>
</html>
```

这里，请求参数总是字符串型，而它需要设置的 index 属性类型为 int 型。此时，JSF 框架会自行完成类型转换。

3.3 值表达式

在 JSF 中，EL(Expression Language)表达式是页面与受管 bean 之间联系的纽带。

EL 表达式的语法格式是 #{expr}，可分为值表达式和方法表达式。通过 EL 值表达式，JSF 页面可以访问 bean 的属性。通过 EL 方法表达式，JSF 页面能够调用 bean 的方法。本节介绍值表达式，第 3.4 节介绍方法表达式。

3.3.1 值表达式的基本用法

值表达式一般出现在页面中，用于访问 bean 的属性。假如页面中有如下 JSF 标记：

```
<h:inputText value="#{user.name}"/>
```

在该标记被呈现时，表达式 user. name 将以右值模式被计算：user 的 getName 方法被调用，name 属性值被读取以便显示。在页面提交时，表达式 user. name 将以左值模式被计算：user 的 setName 方法被调用，name 属性被设置为用户输入的值。

这里，运算符“. ”用于访问 bean 的属性或调用 bean 的方法。在 EL 表达式中，运算符“. ”和“[]”通常可以互相替换使用，或同时混合使用。下面表达式是等价的：

- user. name；
- user["name"]；
- user['name']。

相应地，上面的 JSF 标记可以写成如下：

```
<h:inputText value='#{user["name"]}'/>
```

或

```
<h:inputText value="#{user['name']}"/>
```

实际上，相比运算符“.”，运算符“[]”更加通用化。比如上面访问 name 属性的表达式，也可以采用格式 user[sub_expr]。这里，sub_expr 是一个子表达式，只要它的计算结果为字符串“name”即可。

3.3.2 访问表、映射和数组

值表达式不仅可以访问受管 bean 的简单属性，也可以访问表(List)、映射(Map)或者数组等属性或对象的元素。

假设 t 是一个 List 型对象，则可以用表达式 t[i]访问索引值为 i 的表元素：

- 在右值模式下，方法 t.get(i)被调用，相应的值返回。
- 在左值模式下，方法 t.set(i, value)被调用，相应的值(value)被写入表中。

假设 a 是一个数组对象，则可以用表达式 a[i]访问索引值为 i 的数组元素：

- 在右值模式下，元素 a[i]被访问，相应的值返回。
- 在左值模式下，元素 a[i]被访问，相应的值被写入数组中。

在上面访问表元素或数组元素的表达式中，i 一般应为整数或能够转换成整数的字符串，如 t[3]、a["3"]。更一般地，索引值也可通过子表达式获得。比如，n 是名为 me 的受管 bean 的一个 int 型属性，则可以用表达式 a[user.n]访问数组 a 的一个元素。

假设 m 是一个 Map 型对象，则可以用表达式 m["key"](或 m['key']、m.key)访问映射中键为字符串"key"的对应值：

- 在右值模式下，方法 m.get("key")被调用，相应的值返回。
- 在左值模式下，方法 m.put("key", value)被调用，相应的值(value)被写入映射中。

如果要访问的键—值对的键预先未知，或者键的类型不是 String，则可用更加一般化的格式 m[sub_expr]来访问，即键由其中的子表达式计算获得。

3.3.3 预定义对象及初始项解析

在值表达式中，可以访问系统预定义的对象。表 3-1 列出了访问这些预定义对象的变量。

表 3-1 在 EL 表达式中可用的预定义对象

变 量 名	含 义
header	包含 HTTP 请求头中各域的映射，每个域仅包含 1 个值
headerValues	包含 HTTP 请求头中各域的映射，每个域对应一个 String[]数组
param	包含 HTTP 请求参数的映射，每个请求参数仅包含 1 个值
paramValues	包含 HTTP 请求参数的映射，每个请求参数对应一个 String[]数组
cookie	包含当前请求的 Cookie 名称—值对的映射
initParam	包含该 Web 应用初始化参数的映射

续表

变　量　名	含　　义
requestScope	包含所有请求作用域 bean 或对象的映射
viewScope	包含所有视图作用域 bean 或对象的映射
sessionScope	包含所有会话作用域 bean 或对象的映射
applicationScope	包含所有应用作用域 bean 或对象的映射
flash	包含要转发到下一个视图的对象的映射
resource	包含应用程序资源的映射
facesContext	该请求的 FacesContext 实例对象
view	该请求的 UIViewRoot 实例对象
component	当前组件
cc	当前组合组件

例如，表达式 header["User-Agent"]可以返回请求头中名为 User-Agent 的域的值，即用户浏览器的类型。

对于值表达式 a. b，JSF 一般会按以下步骤解析初始项 a：

（1）是否是一个预定义对象；

（2）依次从请求、视图、会话和应用作用域的映射中寻找；

（3）根据配置文件（典型为 faces-config. xml）中的相关信息或 Bean 类中的 Java 标注信息，寻找是否存在指定名称的受管 Bean。若存在，调用该 Bean 类默认的构造方法创建 bean 实例，并添加至相应作用域的映射中，然后返回该对象。

下面举例说明，假设 JSF 应用中定义了一个名称为 user 的请求作用域的受管 bean，其中包含一个 name 属性。

先考虑表达式 user. name。因为 user 不是一个系统预定义对象，所以 JSF 将依次从下面映射中寻找键为 user 的值，即 bean 实例：

- 请求作用域映射（requestScope）；
- 视图作用域映射（viewScope）；
- 会话作用域映射（sessionScope）；
- 应用作用域映射（applicationScope）。

如果这是作用域内对该受管 bean 的第一次访问，那么这些映射中就不会存在所需的 bean 实例。此时 JSF 将根据配置文件信息和 Java 标注信息，定位 bean 类并创建 bean 实例。由于这是一个请求作用域的受管 bean，因此创建的 bean 实例将作为属性保存在请求作用域映射（requestScope）中。

如果这不是作用域内对该受管 bean 的第一次访问，那么 JSF 将会在请求作用域映射（requestScope）中找到所需的 bean 实例。

再来考虑表达式 requestScope. user. name。因为 requestScope 是一个系统预定义对象，即请求作用域映射，所以 JSF 将在该映射中寻找键为 user 的值，即 bean 实例。如果之

前在作用域内已经访问过该受管 bean，那么 bean 实例就已经存在。此时表达式将返回该 bean 的 name 属性值。如果之前在作用域内还没有访问过该受管 bean，那么该表达式不会返回任何结果。

3.3.4 文字与运算符

在值表达式中，除了能够访问 bean 实例、预定义对象、数组、表和映射及其属性和元素外，还可以包含文字和运算符，实现基本的运算处理。

根据不同的场合，EL 值表达式既可能以右值模式处理，也可能以左值模式处理。但包含文字和运算符的值表达式只能以右值模式处理。

在值表达式中可以使用以下类型的文字。

- 布尔值：true 和 false；
- 整数：如 12、128；
- 浮点数：如 12.6、.5、1.2e6、12E－3；
- 字符串：由单引号或双引号括起来的一串字符，如"computer"、'Liming'；
- 空值：null，表示空的或不存在的引用。

在值表达式中可以使用的运算符如表 3-2 所示。这些运算符与 Java 语言中的运算符大致相同，但其中有些运算符还可以用英文字母来表示。比如，逻辑非既可以用“!”，也可以用 not。

表 3-2 EL 运算符

分　类	运　算　符	优　先　级
访问运算符	[]、.	1
小括号	()	2
单目运算符	－、!(not)、empty	3
算术运算符	＊、/(div)、%(mod)	4
	＋、－	5
关系运算符	＜(lt)、＜＝(le)、＞(gt)、＞＝(ge)	6
	＝＝(eq)、!＝(ne)	7
逻辑运算符	&&(and)	8
	‖(or)	9
条件运算符	? :	10

单目运算符都是前缀运算符，比如 empty a。运算符 empty 用于判断运算对象是否为 null，或是否为空串，或是否为不包含元素的数组、集合或映射。

要注意 null 与 empty 之间的区别。考虑表达式 user.name＝＝null，只有 name 属性值为 null 时，表达式的值才为 true，其他情况表达式的值都为 false。而对于表达式 empty user.name，如果 name 属性值为 null 或空串，表达式的值都将为 true。

在 JSF 中，EL 表达式的主要作用是在应用的表示层和业务层之间架起了一座桥梁，实现了表示逻辑和业务逻辑的分离。在 EL 表达式中进行一些运算，主要是为了实现表示逻

辑，而不应该去完成涉及业务逻辑的复杂的数据处理。例如，下面两个组件标记根据 bean 中 xb 属性值决定是否呈现响应，其中只能有一个组件标记被呈现响应。

```
<h:outputText value="男" rendered="#{user.xb=='1'}" .../>
<h:outputText value="女" rendered="#{user.xb!='1'}" .../>
```

3.3.5 复合表达式

若干值表达式(包括静态文本)可以先后放置组成复合表达式，例如：

```
<h:outputText value="姓名：#{user.name},年龄：#{user.age}"/>
```

这里，各值表达式依其出现的先后次序，从左到右依次计算并转换成字符串，然后和静态文本一起连接形成一个结果字符串。上面标记的呈现结果形如：姓名：李明，年龄：25。

一般来说，复合表达式只能以右值模式处理。

3.4 方法表达式

方法表达式涉及一个对象及其一个 public 方法，其语法与值表达式类似。下面是使用方法表达式的一个例子。

```
<h:commandButton action="#{user.checkPassword}"/>
```

假定 user 代表某受管 bean 实例，checkPassword 是它的一个 public 方法。方法表达式会在某个适当时机计算，其效果是调用对象的指定方法。

在 JSF 中，一个组件标记在服务器端被表示为一个组件对象。用户的操作会抽象成组件对象的事件。通过为组件标记的特定属性设置方法表达式，可以为组件对象的特定事件设置事件处理代码。例如，可以为命令按钮标记的 action 属性设置一个方法表达式，当用户单击命令按钮提交一个请求到服务器时，该方法表达式将被计算，相应的方法将被调用。

有四个组件属性与组件的特定事件相关联，经常需要设置方法表达式。

- action(看第 4.6 节)；
- validator(看第 4.3 节、第 7.5 节)；
- actionListener(看第 4.6 节、第 8.2 节)；
- valueChangeListener(看第 8.3 节)。

由于这些属性设置的方法表达式用于处理不同的事件，所以相应方法的签名会有各自的特点。方法的参数和返回类型取决于使用方法表达式的上下文。比如，action 属性绑定的方法通常要求无形参、返回类型为 String。

```
public String method_name()
```

又比如，actionListener 属性绑定的方法通常需要有一个 ActionEvent 型参数，返回类型为 void。

```
public void method_name(ActionEvent e)
```

这里，开发人员应按上述约定声明对象方法，而 EL 表达式工具以及 JSF 框架将负责提供参

数以及处理返回值。

Java EE 6 平台采用 EL 2.2 版本，支持带参数的方法调用。

```
<表达式>.<标识符>(<parameters>)
```

从语法格式上看，带参数的方法表达式相比值表达式和普通的方法表达式，多了一个参数列表。其中，＜表达式＞的计算结果应该是一个 bean 实例；＜标识符＞代表 bean 实例的一个方法；＜参数列表＞由逗号分隔的多个值或表达式组成。

带参数的方法表达式不见得一定要有参数，但一对圆括号不能省略，这是它与传统的方法表达式的区别。

假设名为 index 的 bean 有如下方法：

```
public String move(int amount){...}
```

那么命令按钮的 action 属性也可设置成如下：

```
<h:commandButton value="Previous" action="#{index.move(-1)}"/>
<h:commandButton value="Next" action="#{index.move(1)}"/>
```

这里，方法重载并不被支持。也就是说，对于方法表达式调用的方法，在相应的 bean 中只能有一个有此方法名的方法。

带参数的方法表达式不仅可以出现在需要方法表达式的属性中，也可以出现在值表达式可以出现的地方。当带参数的方法表达式作为值表达式使用时，它只能以右值模式计算。

3.5　在页面外使用 EL 表达式

大多数情况下，EL 表达式出现在页面中。页面通过 EL 表达式访问受管 bean，实现 JSF 应用的表示层和业务层之间的联系。但有些情况下，也需要在 Faces 配置文件、Java 类中使用 EL 表达式。

3.5.1　通过 EL 表达式初始化受管 bean

可以在 Faces 配置文件中使用 EL 表达式来初始化受管 bean，即可以用 EL 表达式为受管 bean 属性设置初值。这使得一个受管 bean 可以通过引用其他受管 bean 来初始化自身。

这种引用要受到受管 bean 作用域的约束。总体原则是，一个受管 bean 不能引用生命周期比自己短的受管 bean，具体约束情况如表 3-3 所示。

表 3-3　受管 bean 之间的引用受其作用域约束

引用 bean 的作用域	被引用 bean 的作用域
none	none
application	none 和 application
session	none、session 和 application
view	none、view、session 和 application
request	none、requesrt、view、session 和 application

这里，作用域为 none 的受管 bean 可以被任何作用域的受管 bean 引用，但其本身只能引用作用域也为 none 的其他受管 bean。

3.5.2 EL 表达式初始化受管 bean 应用示例

该应用项目(ch3_initializewithel)主要由一个页面和两个受管 bean 组成。项目的运行效果如图 3-3 所示。

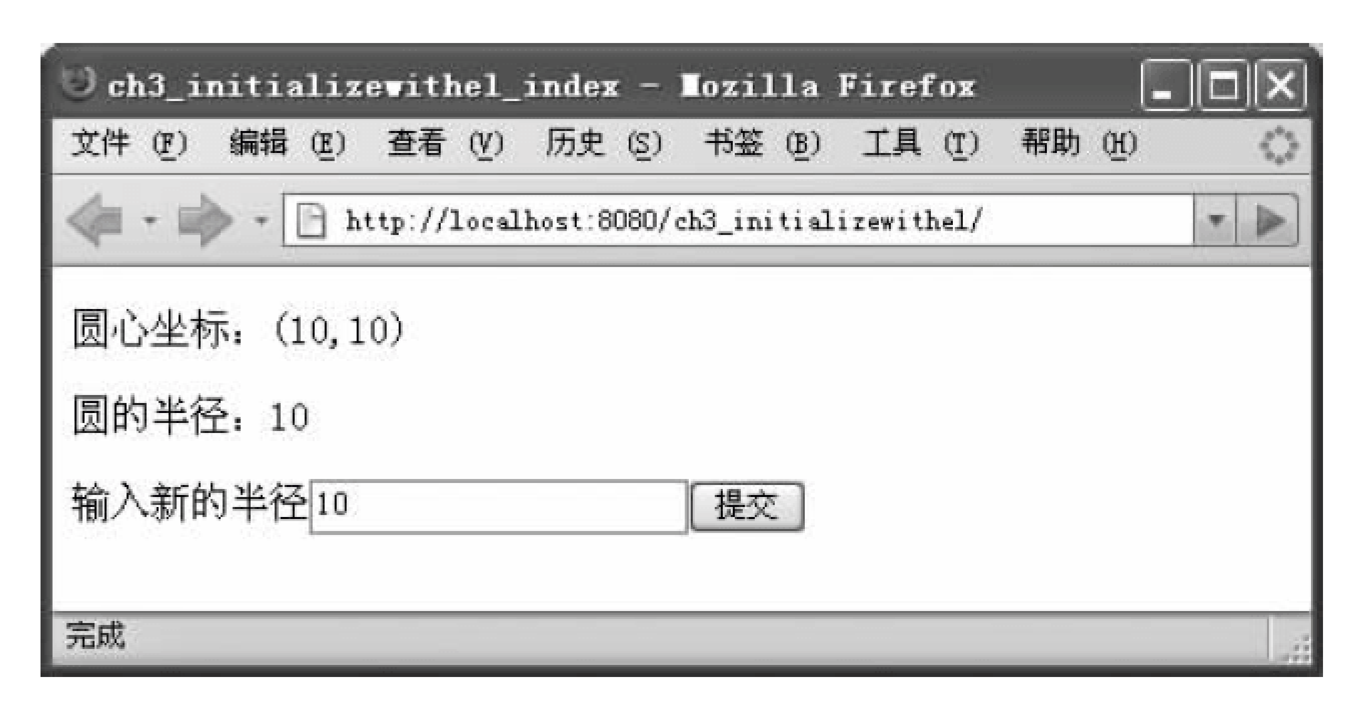


图 3-3 应用 ch3_initializewithel 运行效果图

两个受管 bean 包含一个圆区域的状态数据。页面显示该圆区域的状态，包括圆心的 x 坐标值和 y 坐标值，以及圆的半径。其中的圆的半径可以由用户修改。

1. 创建配置文件

默认情况下，新创建的 JSF 应用项目不包含 Faces 配置文件。本应用中，首先创建配置文件 WEB-INF\faces-config.xml。

2. 创建受管 bean

本应用包含两个受管 bean：Circle.java 和 R_Circle.java。受管 bean 的配置信息不采用 Java 标注形式声明，而是放置在配置文件中。表 3-4 是两个受管 bean 的基本配置信息。

表 3-4 应用 ch3_initializewithel 中受管 bean 的基本配置

bean 类	bean 名称	作 用 域
bean.Circle	circle	request
bean.R_Circle	rc	session

Circle 类表示圆，每个圆对象有一个 radius 属性，表示圆的半径。R_Circle 类表示圆区域，每个圆区域有 3 个属性：x 表示圆心的横坐标，y 表示圆心的纵坐标，c 表示圆对象。Circle 类的定义参看本章一开始的代码清单 3-1，R_Circle 类的定义如代码清单 3-8。

代码清单 3-8 R_Circle.java

```
package bean;
import java.io.Serializable;

public class R_Circle implements Serializable {

    private int x;
```

```
    public int getX(){
      return x;
    }
    public void setX(int x){
      this.x=x;
    }

    private int y;
    public int getY(){
      return y;
    }
    public void setY(int y){
      this.y=y;
    }

    private Circle c;
    public Circle getC(){
      return c;
    }
    public void setC(Circle c){
      this.c=c;
    }
}
```

3. 配置受管 bean

配置文件 faces-config. xml 中有关受管 bean 的配置信息如代码清单 3-9 所示。其中，受管 bean 的类名、bean 名称及作用域等信息在创建受管 bean 时自动添加，而受管 bean 的初始化信息由手工编制完成。

这里，circle 受管 bean 的 radius 属性值初始化为 10，而 rc 受管 bean 的各属性则通过 EL 表达式初始化。其中，x 和 y 都设置为 circle 受管 bean 的 radius 属性值，c 就设置为 circle 受管 bean 本身。

代码清单 3-9　faces-config. xml

```
<managed-bean>
  <managed-bean-name>rc</managed-bean-name>
  <managed-bean-class>bean.R_Circle</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>x</property-name>
    <value>#{circle.radius}</value>
  </managed-property>
  <managed-property>
    <property-name>y</property-name>
    <value>#{circle.radius}</value>
  </managed-property>
  <managed-property>
    <property-name>c</property-name>
```

```
    <value>#{circle}</value>
  </managed-property>
</managed-bean>
<managed-bean>
  <managed-bean-name>circle</managed-bean-name>
  <managed-bean-class>bean.Circle</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>radius</property-name>
    <value>10</value>
  </managed-property>
</managed-bean>
```

4. JSF 页面

本应用仅有一个页面(代码清单 3-10),它首先显示 rc 受管 bean(圆区域)的的圆心坐标及圆半径,然后提供一个表单,该表单可以重新设置 circle 受管 bean(圆)的半径。

代码清单 3-10　index.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>ch3_initializewithel_index</title>
  </h:head>
  <h:body>
    <p>圆心坐标:(#{rc.x},#{rc.y})</p>
    <p>圆的半径:#{rc.c.radius}</p>
    <h:form>
      <h:outputLabel id="l1" for="i1" value="输入新的半径"/>
      <h:inputText id="i1" value="#{circle.radius}"/>
      <h:commandButton value="提交"/>
    </h:form>
  </h:body>
</html>
```

当页面被第一次请求时,请求作用域的 rc 受管 bean 实例和会话作用域的 circle 受管 bean 实例会被依次创建和初始化。当页面呈现响应后,rc 受管 bean 实例被销毁,而 circle 受管 bean 实例仍保持有效。当用户通过表单提交再次请求该页面时,circle 受管 bean 实例的 radius 属性被重新设置,在页面呈现响应时,rc 受管 bean 实例被再次创建并初始化。

3.5.3　在 Java 类中计算 EL 表达式

要在 Java 类(如 bean 类)中访问 EL 表达式,需要涉及两个类。

(1) javax.faces.context.FacesContext。

对每一次 JSF 请求处理,都有一个 FacesContext 对象与之相关联。FacesContext 对象

包含着本次请求处理的所有状态信息。

（2）javax.faces.application.Application。

每一个 JSF 应用都存在一个单一的、能被所有用户共享的 Application 对象，它提供创建和计算 EL 表达式的功能。

要获得与当前其请求相关联的 FacesContext 对象，可以调用 FacesContext 类的以下静态方法。

```
public static FacesContext getCurrentInstance()
```

要获得当前 JSF 应用的 Application 对象，可以调用 FacesContext 对象的以下方法。

```
public Application getApplication()
```

要计算一个 EL 表达式，可以调用 Application 对象的以下方法。

```
public <T>T evaluateExpressionGet(FacesContext context,String expression,
                                   Class<? extends T>expectedType)
```

该方法有 3 个参数。第一个参数需要指定一个 FacesContext 对象。第二个参数是字符串，其内容是需要计算的 EL 表达式。第三个参数以 Class 对象形式指定计算结果的期望类型，该参数也决定了方法的返回类型。方法返回指定 EL 表达式的计算结果。

可以在 Java 类中定义以下静态方法，其功能是返回一个指定名称的 bean 实例。方法的核心代码是计算一个 EL 表达式，表达式的功能是访问指定名称的 bean 实例。如果该 bean 实例已经存在，则直接返回，否则创建并返回该 bean 实例。

```
public static Object getBean(String name){
    FacesContext facesContext=FacesContext.getCurrentInstance();
    Application application=facesContext.getApplication();
    Object bean=application.evaluateExpressionGet(facesContext,"#{"+name+"}",
                Object.class);
    return bean;
}
```

假设上述方法定义在名为 ELUtil 的类中，下面代码是使用上述方法的一个示例。其中，user 是一个受管 bean 的名称，其类名为 UserBean。

```
UserBean bean=(UserBean)ELUtil.getBean("user");
```

一旦获得了受管 bean 实例的引用，就可以像调用普通 Java 对象一样访问该 bean 实例。上述方法代码既可以出现在 bean 类中，也可以用于在普通的 Java 类中。但需要注意的是，它们必须在 JSF 请求处理过程中被调用和执行。

3.6 小　　结

- 受管 bean 是指受 JSF 框架的受管 bean 工具管理的 JavaBean。
- 开发人员负责编写受管 bean 类、配置受管 bean，并可根据需要访问受管 bean 实例。受管 bean 工具根据访问需要和配置情况，负责创建、初始化受管 bean 实例，并维护

受管 bean 实例。

- 受管 bean 的配置信息包括 bean 名称、作用域和属性初值等，可以在 bean 类中用相应的 Java 标注声明，也可以在 Faces 配置文件中用相应的元素声明。
- EL 表达式分为值表达式和方法表达式。值表达式用于访问 bean 的属性，方法表达式用于调用 bean 的方法。
- 页面呈现时，相关的值表达式以右值模式计算。页面提交时，相关的值表达式以左值模式计算。如果一个值表达式，既需要以右值模式计算又需要以左值模式计算，则其访问的 bean 属性必须是可读写的。
- 值表达式中可包含文字和运算符以实现简单运算，此时值表达式只能以右值模式计算。
- 带参数的方法表达式可以作为值表达式使用，此时它只能以右值模式计算。

习　题　3

1. 何谓受管 bean？ 如何声明受管 bean？
2. 受管 bean 的作用域有哪几种？ 简述视图作用域与请求作用域之间的区别。
3. 简述在 bean 类中计算一个 EL 表达式的方法。
4. 假设有如下受管 bean 类：

```
package bean;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class Rectangle {
    private boolean isFill;
    private int width;
    private int height;

    public int getHeight(){
        return height;
    }
    public void setHeight(int height){
        this.height=height;
    }
    public int getWidth(){
        return width;
    }
    public void setWidth(int width){
        this.width=width;
    }
    public int getArea(){
        return width * height;
```

```
    }
}
```

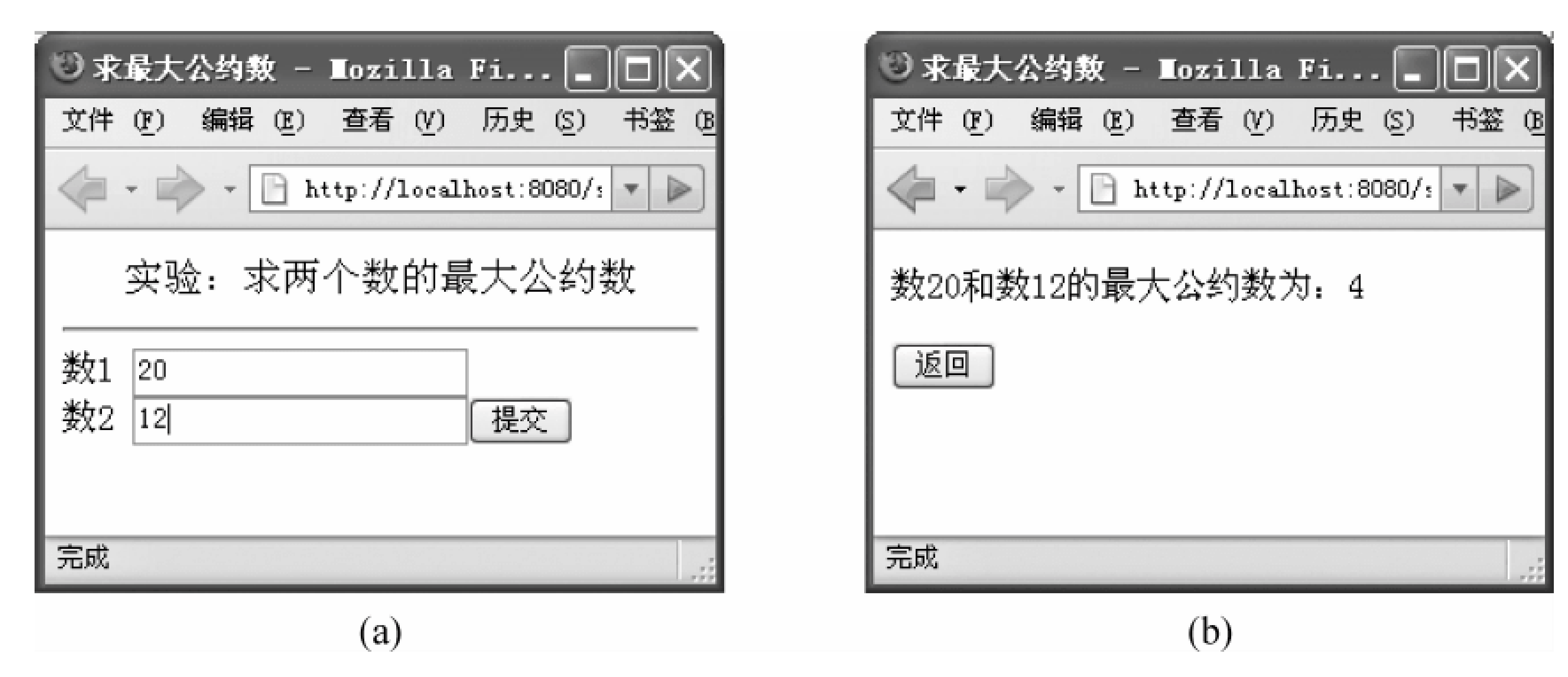
那么该受管 bean 的名称是什么？类名是什么？作用域是什么？可读写属性包括哪些？只读属性包括哪些？

5．假设一个 JSF 页面包含如下表单标记：

```
<h:form id="f">
    <h:inputText id="i" value=""/>
    <h:commandButton value="OK" action="#{user.action}"/>
</h:form>
```

其中，文本域的 value 属性并没有与某受管 bean 的一个属性进行绑定，那么如何在名为 user 的 bean 的 action 方法中获取用户在文本域中输入的值？

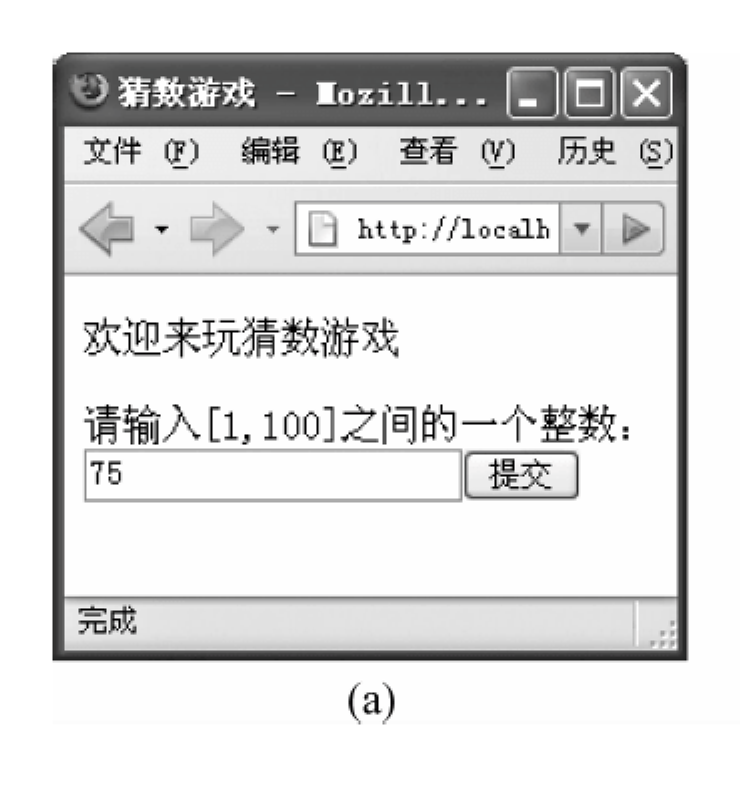
6．编制一个 JSF 应用(sh3_gcd)，可以求两个数的最大公约数。应用包含两个 JSF 页面。第一个页面允许用户输入两个数并提交，如图 3-4(a)所示。第二个页面显示计算结果，单击命令按钮后会返回第一个页面，如图 3-4(b)所示。



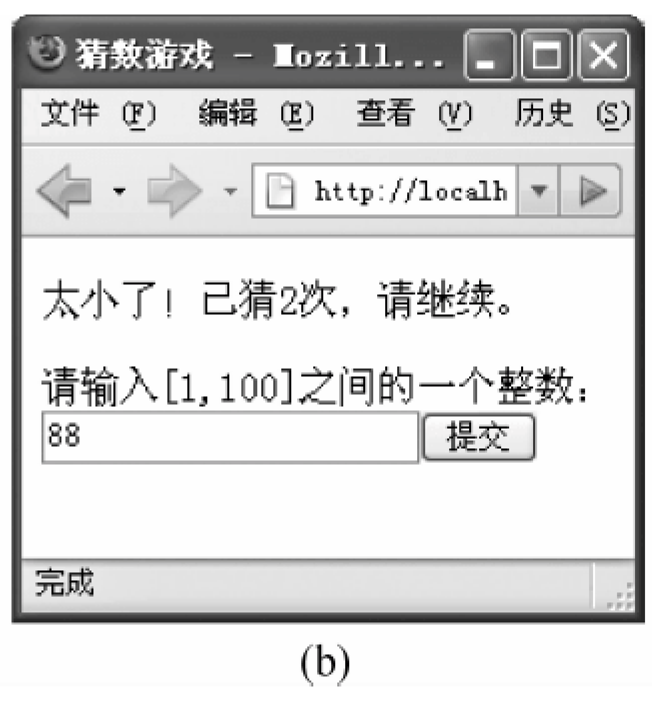
(a) (b)

图 3-4　第 6 题示意图

7．编制一个 JSF 应用(sh3_guess_number)，实现网上猜数游戏。每次游戏开始时，都会自动产生一个 1～100 的随机数，然后显示如图 3-5(a)所示的页面。若用户输入的数相比预先产生的随机数大或小，则显示类似如图 3-5(b)所示的页面。若用户输入的数与预先产生的随机数相等，则显示如图 3-5(c)所示的页面，单击其中的超链接可重新开始游戏。



(a)



(b)

猜数游戏 - Mozill...
文件(F) 编辑(E) 查看(V) 历史(S)
http://localh
成功！共猜了6次。
再来一次
完成

(c)

图 3-5　第 7 题示意图

第 4 章　使用 JSF 标记

本章主题：

- JSF 页面概述
- JSF HTML 标记概述
- 基本输入类标记
- 基本输出类标记
- 图像标记
- 动作类标记
- 二选一标记
- 单选类标记
- 多选类标记
- 消息标记

在 JSF 应用中，用户界面由 JSF 页面表示，JSF 页面主要由 JSF 标记组成。JSF 为 JSF 页面的创建提供了两种可选的技术，一种是基于 JSP 的技术，另一种是 Facelets 技术。相对来说，Facelets 技术为开发人员提供了更为简单易用且功能强大的特性，并成为 JSF 2.0 规范中首选的创建 JSF 页面的技术。本书采用基于 Facelets 技术的 JSF 页面。

在 JSF 标记和 JSF 页面中可以使用 EL 表达式。通过 EL 表达式，页面可以方便地访问受管 bean 属性或调用受管 bean 方法。这是 JSF 页面与普通的静态 Web 页面在功能上的主要区别。基于这一机制，既可以有效地将应用的数据表示与数据处理分离，又可以很好地将它们连接在一起。

要有效地创建 JSF 页面，必须熟练掌握 JSF 标记的使用。本章首先对 JSF 页面和 JSF 标记进行概述，然后详细介绍大多数基本的 JSF HTML 标记的功能和用法。部分 JSF HTML 标记和 JSF 核心标记会在其他一些专门的章节分别介绍。

4.1　JSF 页面概述

JSF 页面由一些不同种类的元素（标记）组成。这里先对组成 JSF 页面的各种标记进行简单描述，然后对 JSF 核心标记做总体介绍。

4.1.1　JSF 页面的组成元素

基于 Facelets 技术的 JSF 页面是一个 XHTML 页面，其文件扩展名为.xhtml。XHTML 是一种符合 XML 语法、良构的、可扩展的 HTML。

在 JSF 页面中，可以使用普通的 HTML 标记，但应满足以下要求：

（1）所有标记必须是闭合的，如以＜p＞为开始标记，以＜/p＞为结束标记。如果是单

独不成对的标记,可以在标记最后加斜杠“/”来关闭它,如
。

(2) 所有的标记名和属性名都必须是小写的。

(3) 所有的属性值应该用单引号或双引号括起来。

这些 HTML 标记不需要经过 JSF 框架的特殊处理,通常被原封不动发往客户端。

一般来说,JSF 页面主要由 JSF 标记组成,另外也可包含一些 JSTL(JSP Standard Tag Library)标记。为了在页面中使用这些 JSF 标记或 JSTL 标记,需要声明相应的 XML 名称空间。表 4-1 列出了 JSF 支持的相关标记库。

表 4-1 JSF 支持的标记库

标 记 库	URI	前缀	例 子
JSF HTML 标记库	http://java.sun.com/jsf/html	h	h:body
JSF 核心标记库	http://java.sun.com/jsf/core	f	f:actionListener
JSF Facelets 标记库	http://java.sun.com/jsf/facelets	ui	ui:component
JSF 复合标记库	http://java.sun.com/jsf/composite	composite	composite:interface
JSTL 核心标记库	http://java.sun.com/jsp/jstl/core	c	c:forEach
JSTL 函数标记库	http://java.sun.com/jsp/jstl/functions	fn	fn:toUpperCase

本章主要介绍 JSF HTML 标记库的标记,JSF 核心标记库的标记将分散在本书各章介绍,JSF Facelets 标记库和复合标记库将在第 10 章介绍。有关 JSTL 标记库的内容,读者可参阅其他图书,本书不做专门介绍。

当然,JSF 页面大多会包含一些 EL 表达式。EL 表达式既可以出现在标记的属性中,也可出现在页面的静态模板文本中。

4.1.2 JSF 核心标记一览

要使用 JSF 核心标记库,需要在页面中声明相应的名称空间:

```
xmlns:f="http://java.sun.com/jsf/core"
```

JSF 核心标记通常执行一些与任何特定的呈现包无关的核心动作。表 4-2 列出了一些常用 JSF 核心标记。

表 4-2 常用的 JSF 核心标记

类 别	标 记	功 能
事件处理	f:actionListener	为父组件添加一个动作监听器
	f:phaseListener	为页面添加一个阶段监听器
	f:valueChangeListener	为父组件添加一个值变化监听器
数据转换	f:converter	为父组件添加一个转换器
	f:convertDateTime	为父组件添加一个 DateTime 转换器
	f:convertNumber	为父组件添加一个 Number 转换器

续表

类　别	标　记	功　能
验证器	f:validator	为父组件添加一个自定义验证器
	f:validateLength	为父组件添加一个 LengthValidator 验证器
	f:validateLongRange	为父组件添加一个 LongRangeValidator 验证器
	f:validateDoubleRange	为父组件添加一个 DoubleRangeValidator 验证器
选项设置	f:selectItem	指定一个选项
	f:selectItems	指定一组选项
参数处理	f:param	参数化消息格式模板或向一个 URL 添加请求参数
视图参数	f:viewParam	为页面视图指定一个视图参数
命名关系	f:facet	为嵌套于它的组件与包含它的组件间建立一种特定关系
本地化	f:loadBundle	装入一个资源包

4.2　JSF HTML 标记概述

本节对 JSF HTML 标记做总体描述,并对 JSF HTML 标记的若干基本属性进行介绍。

4.2.1　JSF HTML 标记一览

要使用 JSF HTML 标记库,需要在页面中声明相应的名称空间:

```
xmlns:h="http://java.sun.com/jsf/html"
```

JSF HTML 标记又称 JSF 组件标记,每个 JSF HTML 标记代表某种 UIComponent 组件与 JSF 框架的 HTML 呈现包中某个呈现器的一种组合。一个 JSF HTML 标记在服务器端表示为 UIComponent 抽象类的某个具体子类的实例对象。在应用请求值阶段,由相应的呈现器获取属于该组件的请求参数并保存到该组件对象内。在呈现响应阶段,由相应的呈现器将组件对象呈现为对应的 HTML 标记。表 4-3 列出了常用的 JSF HTML 标记。

表 4-3　常用的 JSF HTML 标记

分　类	标　记	组　件　类	描　述
表单	h:form	HtmlForm	呈现为 HTML form 元素
头与体	h:head	UIOutput	呈现为 HTML head 元素
	h:body	UIOutput	呈现为 HTML body 元素
基本输入	h:inputText	HtmlInputText	文本域
	h:inputSecret	HtmlInputSecret	口令域
	h:inputTextarea	HtmlInputTextarea	文本区
	h:inputHidden	HtmlInputHidden	隐藏域

续表

分　类	标　　记	组　件　类	描　　述
基本输出	h:outputText	HtmlOutputText	显示单行文本
	h:outputLabel	HtmlOutputLabel	组件的标签
	h:outputLink	HtmlOutputLink	超链接
	h:outputFormat	HtmlOutputFormat	带参数的文本
	h:outputStylesheet	UIOutput	链接外部样式表
图像	h:graphicImage	HtmlGraphicImage	显示图像
动作	h:commandButton	HtmlCommandButton	提交按钮或重置按钮
	h:commandLink	HtmlCommandLink	作用类似于提交按钮的超链接
结果	h:button	HtmlOutcomeTargetButton	结果按钮
	h:link	HtmlOutcomeTargetLink	结果超链接
二选一	h:selectBooleanCheckbox	HtmlSelectBooleanCheckbox	复选框
单选	h:selectOneRadio	HtmlSelectOneRadio	单选按钮组
	h:selectOneMenu	HtmlSelectOneMenu	单选菜单
	h:selectOneListbox	HtmlSelectOneListbox	单选列表框
多选	h:selectManyCheckbox	HtmlSelectManyCheckbox	复选框组
	h:selectManyMenu	HtmlSelectManyMenu	多选菜单
	h:selectManyListbox	HtmlSelectManyListbox	多选列表框
消息	h:message	HtmlMessage	显示组件消息
消息组	h:messages	HtmlMessages	显示全局消息
数据表格	h:dataTable	HtmlDataTable	显示数据表格
列	h:column	HtmlColumn	指定数据表格中的一列
面板	h:panelGrid	HtmlPanelGrid	网格面板
	h:panelGroup	HtmlPanelGroup	组面板

除结果类标记、数据表格标记、列标记、面板类标记和 h:outputStylesheet 标记外，表中其他 JSF HTML 标记都将在本章逐一介绍。结果类标记在第 5 章详细介绍，数据表格标记、列标记、面板类标记和 h:outputStylesheet 标记将在第 6 章介绍。

绝大多数 JSF HTML 标记的标记名都由两部分组成，前半部分是相应组件类的组件族名，后半部分是相应呈现器型名。例如，对于 h:inputText 标记，其组件族名为 Input，其呈现器型名为 Text。

说明：组件族名和呈现器型名都具有前缀 javax.faces。所以上述组件族的完整名称应该是 javax.faces.Input，呈现器型名的完整名称应该是 javax.faces.Text。

有些标记名并不符合上述约定：

（1）对于 h:head 标记，相应的组件族名为 Output，相应的呈现器型名为 Head。

（2）对于 h:body 标记，相应的组件族名为 Output，相应的呈现器型名为 Body。

（3）对于 h:form 标记，相应的组件族名和呈现器型名均为 Form。

（4）对于 h:column 标记，相应的组件族名为 Column，但无呈现器型名。h:column 标记一般嵌套于 h:dataTable 标记，由后者相应的呈现器统一呈现。

每个 JSF HTML 标记组件都有组件族名和呈现器型名两个属性，这两个属性共同决定该组件相关的呈现器类型。下面代码根据组件族名和呈现器型名返回一个呈现器：

```
RenderKit kit=FacesContext.getCurrentInstance().getRenderKit();
Renderer renderer=kit.getRenderer("javax.faces.Input", "javax.faces.Text");
```

进一步，同一种呈现器也可能服务于不同类型的组件，此时，呈现器的行为也会因组件类型不同而不同。

h:head 标记呈现为 HTML head 标记，h:body 标记呈现为 HTML body 标记，h:form 标记呈现为 HTML form 标记。

h:form 可以包含一组子组件标记，并在呈现时形成一个表单。一个表单可以收集用户数据，这些数据将作为请求参数随 HTTP 请求送往服务器。一个 JSF 页可以包含多个表单，但只有被客户提交的表单的数据会随请求送往服务器。

表 4-3 中各标记是按其相应组件的组件族分类的，即同一组各标记相应组件具有相同的组件族名。例如，基本输入类中各标记相应组件的组件族名为 Input，单选类中各标记相应组件的组件族名为 SelectOne。具有相同组件族名的组件类往往具有共同的超类，该超类名为 UI 加组件族名。例如，基本输入类中各组件类的共同超类为 UIInput，单选类中各组件类的共同超类为 UISelectOne。当然，所有的组件类都是 UIComponent 抽象类的子类。

这里，名称以 UI 开头的组件类属于 javax. faces. component 包，名称以 Html 开头的组件类属于 javax. faces. component. html 包。

4.2.2 基本属性

下面是一些基本的、大多数 JSF HTML 标记都具有的属性。

1. id

该属性用于指定组件标识符。组件标识符应以字母打头，后续字符可以是字母、数字、下划线“_”或破折号“—”。若不指定该属性，JSF 框架会在需要时为组件自动产生标识符。

在页面中，一个标记可以通过组件标识符引用另一个标记。例如，可以按下面方式为一个输入标记设置一个标签：

```
<h:form id="form1">
    <h:outputLabel for="input1" value="请输入"/>
    <h:inputText id="input1" value="123"/>
    <h:commandButton value="OK"/>
</h:form>
```

第 2.2.3 小节曾经介绍，一个组件既有组件标识符，又有客户端标识符。在一个页面视图内，各组件的组件标识符不一定互不相同，但其客户端标识符总是唯一的。所以在服务器

端 Java 代码中，可以通过组件的客户端标识符唯一标识一个组件对象：

```
//获取组件树的根
UIViewRoot root=FacesContext.getCurrentInstance().getViewRoot();
//返回指定客户端标识符的组件对象
UIComponent component=root.findComponent("form1:input1");
```

2. immediate

输入类组件和动作类组件通常可以设置 immediate 属性。该属性的默认值为 false。若将其设置为 true，则输入类组件数据值的转换、验证或者动作类组件动作事件的处理将会在应用请求值阶段直接执行，而不会等到验证处理、调用应用等阶段再执行。

该属性设置为 true 的组件也称为直接组件。第 8.3.4 小节和第 8.5.4 小节所举的两个示例都用到了该属性。

3. rendered

该属性决定组件是否被 JSF 框架呈现输出。默认值为 true，即组件会被呈现、产生相应的 HTML 标记。经常地，可以将该属性设置为一个 boolean 型 EL 表达式，从而能根据表达式的计算结果，动态决定一个组件是否被呈现输出。

4. style 和 styleClass

这两个属性为组件的呈现输出指定 CSS(Cascading Style Sheets)样式。style 属性指定 CSS 样式，并作为呈现产生的 HTML 标记的 style 属性值；styleClass 属性指定 CSS 样式类，并作为呈现产生的 HTML 标记的 class 属性值。

下面代码演示了这两个属性的使用。output1 组件使用 style 属性直接指定 CSS 样式；output2 组件在 styleClass 属性中指定了一个样式类，该样式类定义于页的头部。

```
<h:head>
    <title>style 与 styleClass 属性</title>
    <style type="text/css">
        .s1 {
            font-size: 24px
        }
    </style>
</h:head>
<h:body>
    <h:outputText id="output1" style="color: red" value="style 属性的使用"/>
    <br/>
    <h:outputText id="output2" styleClass="s1" value="styleClass 属性的使用"/>
</h:body>
```

在呈现响应阶段，output1 组件和 output2 组件将依次产生如下输出：

```
<span id="output1" style="color: red">style 属性的使用</span>
<span id="output2" class="s1">styleClass 属性的使用</span>
```

5. value 和 binding

value 属性用于设置组件值。该属性可设置为普通的字符串文本，但经常设置为 EL 值

表达式，使组件值与受管 bean 的某个属性绑定在一起。

binding 属性总是设置为 EL 值表达式，它使组件本身与受管 bean 的某个属性绑定在一起。EL 值表达式的类型必须与标记代表的组件类型相容。

下面代码演示了 binding 属性的使用。binding 属性指定 cj 组件对象与名称为 student 的受管 bean 中的 score 属性绑定在一起。也就是说，score 属性引用 cj 组件对象。

```
<h:inputText id="cj" binding="#{student.score}"/>
```

这样，就可以在 bean 类中通过 score 属性对 cj 组件进行各种操作。当然，首先需要 bean 类中对 score 属性进行声明。通常可以将 score 属性的类型声明为 UIInput，如果需要对 cj 组件进行与呈现相关的属性的操作，则应用将其类型声明为 HtmlInputText。

6．title

为该组件呈现产生的标记元素指定标题信息。Web 浏览器通常会将该信息以即时提示的方式显示出来。

4.3　基本输入类标记

基本输入类标记的相应组件类有共同的超类 UIInput，在该超类中定义了这些组件共同的组件族名 Input。

4.3.1　标记功能

基本输入类标记共四个。

1．h:inputText

文本域，用于接收一行文本。

在服务器端表示为一个 HtmlInputText 组件实例。组件呈现为 HTML input 元素，type 属性设置为“text”。

2．h:inputSecret

口令域，与文本域一样用于接收一行文本，但在输入时仅显示为一串 *，而不会显示出用户实际输入的内容。

在服务器端表示为一个 HtmlInputSecret 组件实例。组件呈现为 HTML input 元素，type 属性设置为“password”。

3．h:inputTextarea

文本区，用于接收多行文本。

在服务器端表示为一个 HtmlInputTextarea 组件实例。组件呈现为 HTML textarea 元素。

4．h:inputHidden

隐藏域，允许在表单中隐藏一个数据，以便在表单提交时，该数据能送回给服务器。

在服务器端表示为一个 HtmlInputHidden 组件实例。组件呈现为 HTML input 元素，type 属性设置为“hidden”。

4.3.2 常用属性

下面介绍基本输入类标记常用的一些属性。有些属性可能仅适用于某些标记，而不适用于其他标记。有些标记可能既适用基本输入类标记，也适用于其他 JSF HTML 标记。

(1) size：用于指定文本域或口令域的显示宽度(列数)。

(2) maxlength：用于指定文本域或口令域最多可以输入的字符数。该属性最终会成为呈现产生的 HTML 标记的同名属性，浏览器据此控制用户输入的字符数。在服务器端，利用代码设置组件值时并不受此属性值的限制。

(3) cols、rows：用于指定文本区的宽度(列数)和高度(行数)。

(4) disabled：指定呈现产生的 HTML 元素是否禁用，默认值为 false。若设置为 true，呈现产生的 HTML 元素将包含属性 disableed＝"disabled"。此时，HTML 元素为禁用状态，不会聚焦。

该属性适用于除隐藏域标记外的其他基本输入类标记，也适用于选择等输入标记、动作标记、h:link 标记和 h:outputLink 标记。

(5) label：指定组件的表现名。当显示有关组件的错误消息时，JSF 框架会用该名称代替组件客户端标识符来标识组件，使错误消息的可读性更强。

该属性不适用于隐藏域标记。

(6) accesskey：指定呈现产生的 HTML 组件的访问键，当用户按下＜ALT＞＋＜访问键＞时将激活当前组件。

该属性适用于除隐藏域标记外的其他基本输入类标记，也适用于选择等输入标记、动作标记、h:outputLink 标记以及 h:outputLabel 标记。

当 h:outputLabel 标记指定了访问键后，用户按下访问键时，将激活与该标签相关联的组件，如聚焦文本域、选中复选框。

(7) readonly：指定呈现产生的 HTML 组件是否只读，默认值为 false。若设置为 true，呈现产生的 HTML 元素将包含属性 readonly＝"readonly"。

该属性适用于除隐藏域标记外的其他基本输入类标记，也适用于选择等输入标记。

说明：比较 readonly 和 disabled 属性，readonly 组件的值会作为请求参数送往服务器，而 disabled 组件的值不会送往服务器。但 JSF 框架并不对 readonly 组件请求值做任何处理。

(8) redisplay：指定在呈现时是否包含组件值，仅适用于口令域标记，默认值为 false。

(9) required：指定用户是否必须为该输入组件提供值，boolean 型。属性值为 true 的输入组件也称为必填项。

(10) requiredMessage：指定必填错误消息，该消息将代替由 JSF 框架产生的错误消息，在用户没有为必填项提供值时显示。

(11) validator：指定一个方法表达式，引用受管 bean 的一个方法。该方法称为验证方法，通常在处理验证阶段被调用。验证方法的方法签名如下：

```
public void <方法名>(FacesContext,UIComponent,Object)
```

(12) validatorMessage：指定验证错误消息，该消息将代替由任何验证器产生的错误消

息，在验证出错时显示。

以上(9)～(12)这4个属性不仅适用于基本输入类标记，也适用于选择等输入标记。

4.4 基本输出类标记

基本输出类标记的相应组件类有共同的超类 UIOutput，在该超类中定义了这些组件共同的组件族名 Output。

4.4.1 标记功能

基本输出类标记包括4个。

1. h:outputText

显示由 value 属性指定的文本，需要时可用 style 和 styleClass 属性指定显示格式。

在服务器端表示为一个 HtmlOutputText 组件实例。当包含 id、style 和 styleClass 等属性时，组件呈现为包含文本的 HTML span 元素；否则直接呈现文本。

2. h:outputLabel

显示一串文本作为某输入类组件(或动作类组件)的标签，由 for 属性指定相关联的输入组件。当单击标签时，会聚焦相关的输入组件。作为标签的文本可以由 value 属性指定，也可以是嵌套于标记内的文本、或由嵌套于标记内的 h:outputText 子标记指定。

该标记在服务器端表示为一个 HtmlOutputLabel 组件实例。组件呈现为 HTML label 元素。

3. h:outputFormat

显示参数化文本。标记的 value 指定一个称为消息模板的特殊字符串，其中由花括号括起来的数字代表一个参数，数字表示参数编号。各参数值则可由嵌套于该标记的各 JSF 核心标记 f:param 的 value 属性依次指定。

下面代码演示了该标记的使用。呈现时，参数{0}替换为 LiMing，参数{1}替换为19。

```
<h:outputFormat value="{0}今年{1}岁,{0}是学计算机的学生。"/>
    <f:param value="LiMing"/>
    <f:param value="19"/>
</h:outputFormat>
```

该标记在服务器端表示为一个 HtmlOutputFormat 组件实例。当包含 id、style 和 styleClass 等属性时，组件呈现为包含参数化文本的 HTML span 元素；否则直接呈现参数化文本。

4. h:outputLink

超链接。value 属性指定超链接的目标资源的 URL。超链接的文本(图像)由嵌套于该标记的静态文本或其他标记指定，如 h:outputText、h:graphicImage 等。

目标资源的 URL 可以是绝对 URL 或相对 URL。对于相对 URL，如果不以“/”开头，则相对于当前页面所在的路径；如果以“/”开头，则相对于当前 Web 容器根目录。例如，单击下面标记产生的超链接，可以导航至当前 Web 容器中上下文路径为“/jsfproject”的 JSF 应用的 response.xhtml 页面：

```
<h:outputLink value="/jsfproject/faces/response.xhtml">
    超链接文本
</h:outputLink>
```

与动作类标记不同，单击超链接不是产生回送(postback)请求，而是产生直接请求。可以在标记中嵌入 JSF 核心标记 f:param 提供请求参数，请参看第 4.6.3 小节中的应用示例。

该标记在服务器端表示为一个 HtmlOutputLink 组件实例。组件呈现为 HTML a 元素。

4.4.2 常用属性

基本输出类标记常用的属性包括 id、value、style、styleClass 等基本属性，其中 value 属性在 h:outputLink 标记中有不同于在其他标记中的作用。除此之外，基本输出类组件还经常用到以下属性：

(1) escape：指定是否对输出内容中的特殊字符(如<、>、&)进行转义处理，boolean 型。默认值为 true，此时特殊字符被转义，如“<”转换成“<”、“>”转换成“>”、“&”转换成“&”，结果它们会按原样在显示器上显示出来。若设置为 false，组件值不被转义处理，特殊字符将会被浏览器解释。

该属性适用于 h:outputText、h:outputLabel 和 h:outputFormat 标记。

(2) for：指定以本标记为标签的输入类组件或动作类组件的客户端标识符。若本标签标记与关联的组件标记处于同一命名容器组件内，可简单指定关联组件的标识符。

该属性仅适用于 h:outputLabel 标记。

(3) target：指定目标资源打开的位置。默认情况下，目标资源会在当前窗口打开。如果将该属性设置为“_blank”，目标资源将在新的窗口或标签页内打开。也可为该属性设置一个名称，那么目标资源将在指定名称的窗口或标签页内打开。

该属性适用于 h:outputLink 标记，也适用于 h:form、h:commandLink 和 h:link 标记。

4.5 图像标记

图像标记 h:graphicImage 用于显示一幅图像，属性 url(或 value)指定图像的 URL。属性 width 和 height 指定图像显示的宽度和高度。

图像的 URL 可以是绝对 URL 或相对 URL。对于相对 URL，如果不以“/”开头，则相对于当前页面所在的路径；如果以“/”开头，则相对于当前 JSF 应用的上下文路径。例如，下面标记可以显示当前 JSF 应用中 image 目录下的 tuxiang.jpg 图像：

```
<h:graphicImage url="/image/tuxiang.jpg"/>
```

该标记在服务器端表示为一个 HtmlGraphicImage 组件实例。组件呈现为 HTML img 元素，其 src 属性等于组件的 url(或 value)属性。如果应用使用 URL 重写来维护会话，组件将自动重写 URL 以维护会话。

4.6　动作类标记

动作类标记的相应组件类有共同的超类 UICommand，在该超类中定义了这些组件共同的组件族名 Command。

4.6.1　标记功能

动作类标记有两个。

1．h：commandButton

动作按钮。type 属性指定按钮的类型，若类型为 submit（默认），产生提交按钮，单击按钮会产生回送（postback）请求、引发相应组件的动作事件；若类型为 reset，单击按钮不会产生 HTTP 请求，只是将表单恢复为填写前的状态，以便重新填写，常称重置按钮。在上述两种情况下，value 属性值作为按钮的标题。动作按钮主要是指提交按钮。

若为 image 属性设置一个图像的 URL（绝对或相对），则产生一个图形化的提交按钮。此时，type 和 value 属性被忽略。

该标记必须放置在 h：form 标记内。

该标记在服务器端表示为一个 HtmlCommandButton 组件实例。组件呈现为 HTML input 元素，其 type 属性等于组件的 type 属性，value 属性等于组件的 value 属性。如果组件包含 image 属性，则 input 元素的 type 属性设置为"image"，src 属性设置为组件的 image 属性。

2．h：commandLink

动作超链接。在外观表现上，它与 h：outputLink 标记类似，也产生一个超链接元素。嵌套的 h：outputText 标记和 h：graphicImage 标记都可以指定超链接的文本和图像。但不同的是，动作超链接不需要指定目标资源，所以它可以用自身的 value 属性指定超链接文本。

在功能行为上，动作超链接与动作按钮相同，即用户单击时提交表单、引发相应组件的动作事件。与动作按钮标记一样，动作超链接标记也应该置于 h：form 标记内。

该标记在服务器端表示为一个 HtmlCommandLink 组件实例。组件呈现为 HTML a 元素，其 href 属性值为"#"，onclick 属性包含一段 JavaScript 代码，可以产生提交表单、发送回送请求的功能行为。

4.6.2　常用属性

动作类标记的功能是提交表单、引发动作事件，所以其最重要的属性莫过于涉及动作事件处理的两个属性。

（1）actionListener：指定一个方法表达式，引用受管 bean 的一个方法。该方法称为动作监听方法，会在组件引发动作事件时被自动调用，通常在"调用应用"阶段被调用。动作监听方法的方法签名如下：

```
public void <方法名> (ActionEvent)
```

或

```
public void <方法名> ()
```

(2) action：该属性在前面已被多次使用。它既可以指定一个字符串文本，也可指定一个方法表达式。方法表达式引用的方法称为动作方法，会在组件引发动作事件时被自动调用，通常在“调用应用”阶段被调用。动作方法的方法签名如下：

```
public String <方法名>()
```

或

```
public Object <方法名>()
```

如果同时指定了动作监听方法和动作方法，则先执行动作监听方法，后执行动作方法。action 属性指定的字符串或其指定的动作方法的执行结果，将作为导航的依据。这里，如果动作方法的返回类型是 Object，那么 JSF 框架会先调用返回结果的 toString()方法将其转换成字符串。有关导航处理的内容在第 5 章做详细介绍。

(3) target：指定响应内容显示的位置。默认情况下，响应内容会在当前窗口显示。如果将该属性设置为“_blank”，响应内容将在一个新打开的窗口或标签页内显示。也可为该属性指定一个名称，那么响应内容将在指定名称的窗口或标签页内显示。

该属性适用于 h:commandLink 标记，也适用于 h:form、h:outputLink 和 h:link 标记。

h:commandButton 标记没有 target 属性，但可以在 h:form 标记中设置 target="_blank"，这样表单内所有的动作按钮或动作超链接提交表单后产生的响应内容都将在新的窗口或标签页中显示。

4.6.3 超链接与动作超链接标记应用示例

该应用项目(ch4_link)主要由两个页面和一个受管 bean 组成。项目的运行效果如图 4-1 所示。

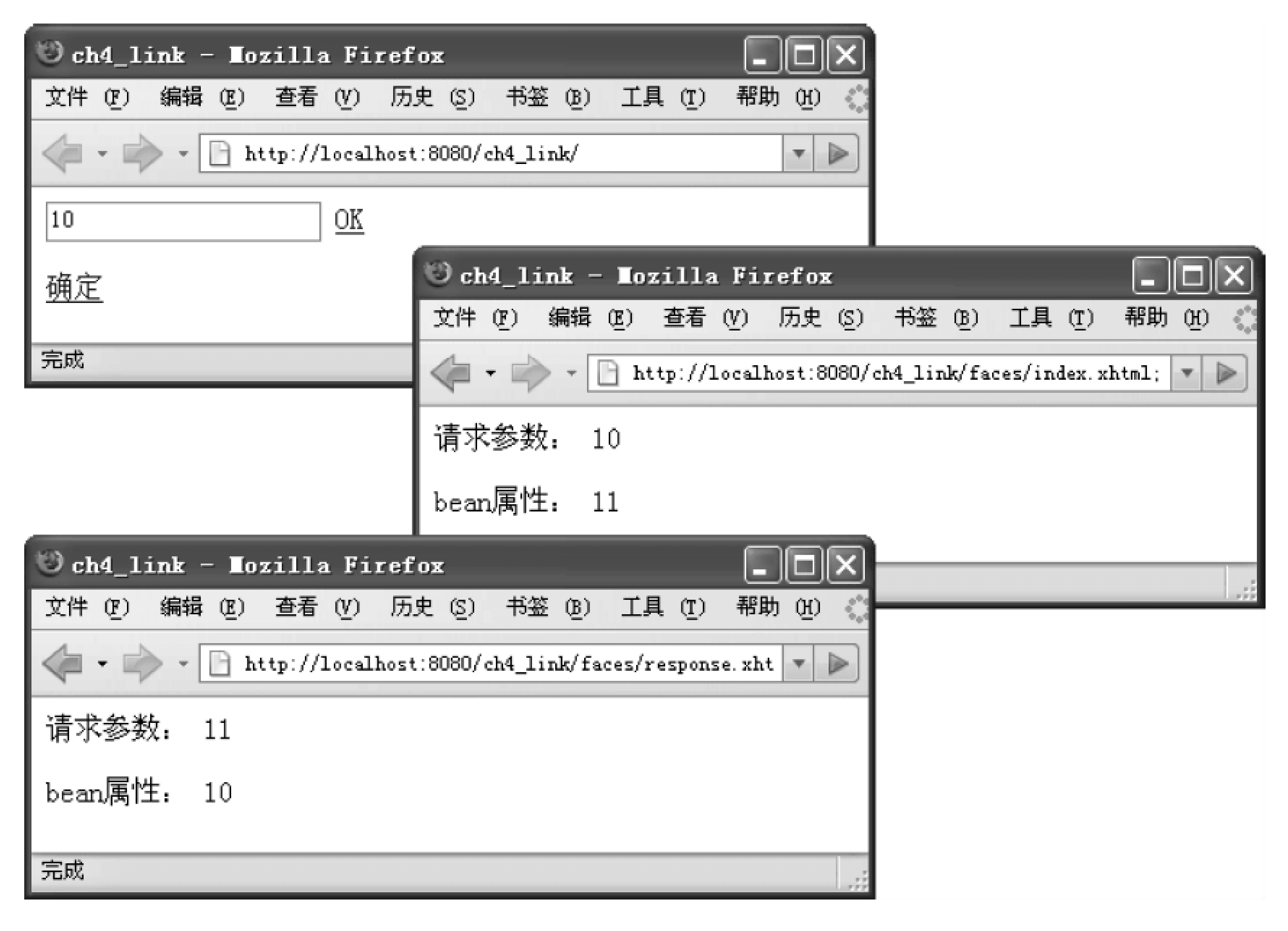


图 4-1　应用 ch4_link 运行效果图

1. 受管 bean

Index.java 是应用唯一的一个受管 bean(代码清单 4-1),其名称为 me、作用域为请求。bean 中定义有一个 age 属性,以及一个 action 方法。

代码清单 4-1　受管 Bean(Index.java)

```
package bean;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean(name="me")
@RequestScoped
public class Index {

  private int age=10;
  public int getAge(){
    return age;
  }
  public void setAge(int age){
    this.age=age;
  }

  public String action(){
    age++;
    return "response";
  }
}
```

2. JSF 页面

应用包含两个页面:欢迎页面 index.xhtml 和响应页面 response.xhtml。

index.xhtml(代码清单 4-2)页面有一个表单,表单中包含一个文本域和一个动作超链接"OK"。文本域中初始显示受管 bean 中 age 属性的值,但用户可以输入新的值。当用户单击动作超链接"OK"时,将产生一个 POST 请求,用户在文本域输入的数据作为一个请求参数一并提交。服务器接收该请求后,将经历恢复视图、应用请求值、处理验证等各阶段。在"更新模型"阶段,用户在文本域中输入的数据被保存到受管 bean 的 age 属性中。在"调用应用"阶段的最后,动作超链接标记的 action 属性指定的受管 bean 的 action 方法被执行,结果 age 属性的值被加 1,然后由导航处理器导航至响应页面。

index.xhtml 页面还包含一个由 h:outputLink 标记产生的普通超链接。该标记无论是出现在 h:form 标记内还是出现在 h:form 标记外,其效果是一样的,即都不会提交表单。其子标记 f:param 指定一个请求参数。单击该超链接将产生一个 GET 请求,其请求参数作为 URL 的一部分会出现在地址栏中。在这里,请求的目标页面的 URL 以及请求参数在本页面呈现时就已经完全确定。

代码清单 4-2　index.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>ch4_link</title>
  </h:head>
  <h:body>
    <h:form id="f1">
      <h:inputText id="i1" value="#{me.age}"/>
      <h:commandLink id="c1" value="OK" action="#{me.action}"/>
      <p></p>
      <h:outputLink value="/ch4_link/faces/response.xhtml">
        确定
        <f:param name="f1:i1" value="#{me.age+1}"/>
      </h:outputLink>
    </h:form>
  </h:body>
</html>
```

response.xhtml(代码清单 4-3)是一个响应页面,通过两个 EL 表达式依次获得和显示请求参数 f1:i1 的值和受管 bean 中 age 属性的值。

代码清单 4-3　response.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>ch4_link</title>
  </h:head>
  <h:body>
    请求参数:
    <h:outputText value="#{param['f1:i1']}"/>
    <p/>
    bean 属性:
    <h:outputText value="#{me.age}"/>
  </h:body>
</html>
```

4.7　二选一标记

二选一标记 h:selectBooleanCheckbox 用于显示一个复选框,用户可以在两种状态中选择其一。该标记的 value 属性通常绑定到受管 bean 的一个布尔型属性上。

例如，一个表单中包含以下两个标记：前者呈现为一个复选框，后者作为该复选框的标签。

```
<h:selectBooleanCheckbox id="bc" value="#{index.accept}"/>
<h:outputLabel for="bc" value="接受条款?"/>
```

其中 h:selectBooleanCheckbox 标记的 value 属性绑定在名称为 index 的受管 bean 的 accept 属性上。accept 属性的类型应该是 boolean 或 Boolean。表单提交时，用户对该复选框上的选择就会保存在 accept 属性中。

该标记在服务器端表示为一个 HtmlSelectBooleanCheckbox 组件实例。组件呈现为 HTML input 元素，其中 type 属性设置为“checkbox”。

4.8 单选类标记

单选类标记的相应组件类有共同的超类 UISelectOne，在该超类中定义了这些组件共同的组件族名 SelectOne。

4.8.1 标记功能

所有单选标记都由子标记 f:selectitem 或 f:selectItems 指定选项。每个选项有一个标签和一个值，标签显示在界面上。对所有单选标记，用户只能选择一项，当选择某个选项时，该选项的值即为单选标记组件的值。单选类标记共 3 个。

1. h:selectOneRadio

单选按钮组，单击按钮图标或标签进行选择。

该标记在服务器端表示为一个 HtmlSelectOneRadio 组件实例。呈现时，由子标记指定的每个选项被呈现为一个 HTML input 元素，type 属性设置为“radio”，name 属性为组件的客户端标识符，value 属性为选项值。每个 input 元素都有各自的 id。每个按钮选项的标签嵌套于 HTML label 元素中，其 for 属性指向对应的 input 元素。

2. h:selectOneListbox

单选列表框。有一个 size 属性，用于指定显示的选项数，单击滚动条或下拉按钮可显示其他选项；若不指定该属性，所有选项将一同显示。

在服务器端表示为一个 HtmlSelectOneListbox 组件实例。组件呈现为 HTML select 元素，其 size 属性设置为组件的 size 属性。由子标记指定的每个选项被呈现为一个 HTML option 子元素。

3. h:selectOneMenu

单选菜单(下拉菜单)。没有 size 属性，一次仅显示一个选项，单击下拉按钮可显示所有选项。

在服务器端表示为一个 HtmlSelectOneMenu 组件实例。组件呈现为 HTML select 元素，其 size 属性设置为 1。由子标记指定的每个选项被呈现为一个 HTML option 子元素。

4.8.2 常用属性

除了一些基本属性(如 id、value 等)，单选类标记常用的属性有以下几种。

(1) border：指定边框宽度，int 型(以像素为单位)。既适用于单选按钮组标记，也适用于复选框组标记(h:selectManyCheckbox)。

(2) enabledClass：指定可用选项标签的 CSS 样式类，String 型。既适用于单选类标记，也适用于多选类标记。

(3) disabledClass：指定不可用选项标签的 CSS 样式类，String 型。与 enabledClass 属性一样，适用于所有的单选类标记和多选类标记。

有关可用选项和不可用选项的含义在 4.8.3 小节“选项设置”中介绍。

(4) hideNoSelectionOption：指定当组件被用户激活时，“非选择项”是否被隐藏，boolean 型。适用于所有的单选类标记和多选类标记。

有关“非选择项”的含义在 4.8.3 小节“选项设置”中介绍。

(5) layout：指定单选按钮组中的单选按钮、复选框组中的复选框等的呈现方向，String 型。有效值包括：

- lineDirection(默认值，水平方向)。
- pageDirection(垂直方向)。

该属性既适用于单选按钮组标记和复选框组标记(h:selectManyCheckbox)，也适用于消息组标记(h:messages)和组面板标记(h:panelGroup)。

4.8.3 选项设置

无论是单选类标记还是多选类标记，其选项都由 f:selectItem 和 f:selectItems 子标记提供。

1. f:selectItem

该标记应嵌套于单选或多选标记内，为选择标记指定单个选项。其主要属性有以下几种。

- itemLabel：指定选项的标签(显示的文字)，String 型。
- itemValue：指定选项值，Object 型。当用户选择选项时，该值作为请求参数送往服务器。
- itemDisabled：指定选项是否禁用，Boolean 型。不是所有的浏览器都支持。
- noSelectionOption：指定该选项是否为“非选择项”，Boolean 型。缺省值是 false。
- value：值表达式，指向一个 SelectItem 实例。此时，选项的各项数据由该 SelectItem 实例提供。

说明：若指定了 value 属性，那么 itemLabel、itemValue 等属性被忽略。

2. f:selectItems

该标记应嵌套于单选或多选标记内，为选择标记指定一组选项。在选择标记内，可以同时存在 f:selectItem 标记和 f:selectItems 标记。

通常情况下，f:selectItems 标记由 value 属性指定一组选项。value 属性必须是指向下面对象之一的值表达式：

- 单个 SelectItem 实例。
- SelectItem 实例集合。
- SelectItem 实例数组。
- 映射(Map 对象)。

当为映射时，映射中的每个键—值对的键和值分别代表一个选项的标签和值。其他情况下，每个 SelectItem 实例确定一个选项的各项数据。

SelectItem 类位于 javax.faces.model 包，下面是该类软件接口的一个摘要。

```
public class SelectItem implements Serializable {
    SelectItem();                                   //构造方法
    SelectItem(Object value,String label);          //构造方法
    public String getLabel();                       //选项标签
    public void setLabel(String label);
    public Object getValue();                       //选项值
    public void setValue(Object value);
    public boolean isDisabled();                    //选项是否禁用
    public void setDisabled(boolean disabled);
    public boolean isNoSelectionOption();           //是否为“非选择项”
    public void setNoSelectionOption(boolean noSelectionOption)
}
```

4.8.4　单选标记应用示例

该应用项目(ch4_calorie)主要由两个页面和一个受管 bean 组成，可以计算一个人一天需要消耗的能量。项目的运行效果如图 4-2 和图 4-3 所示。

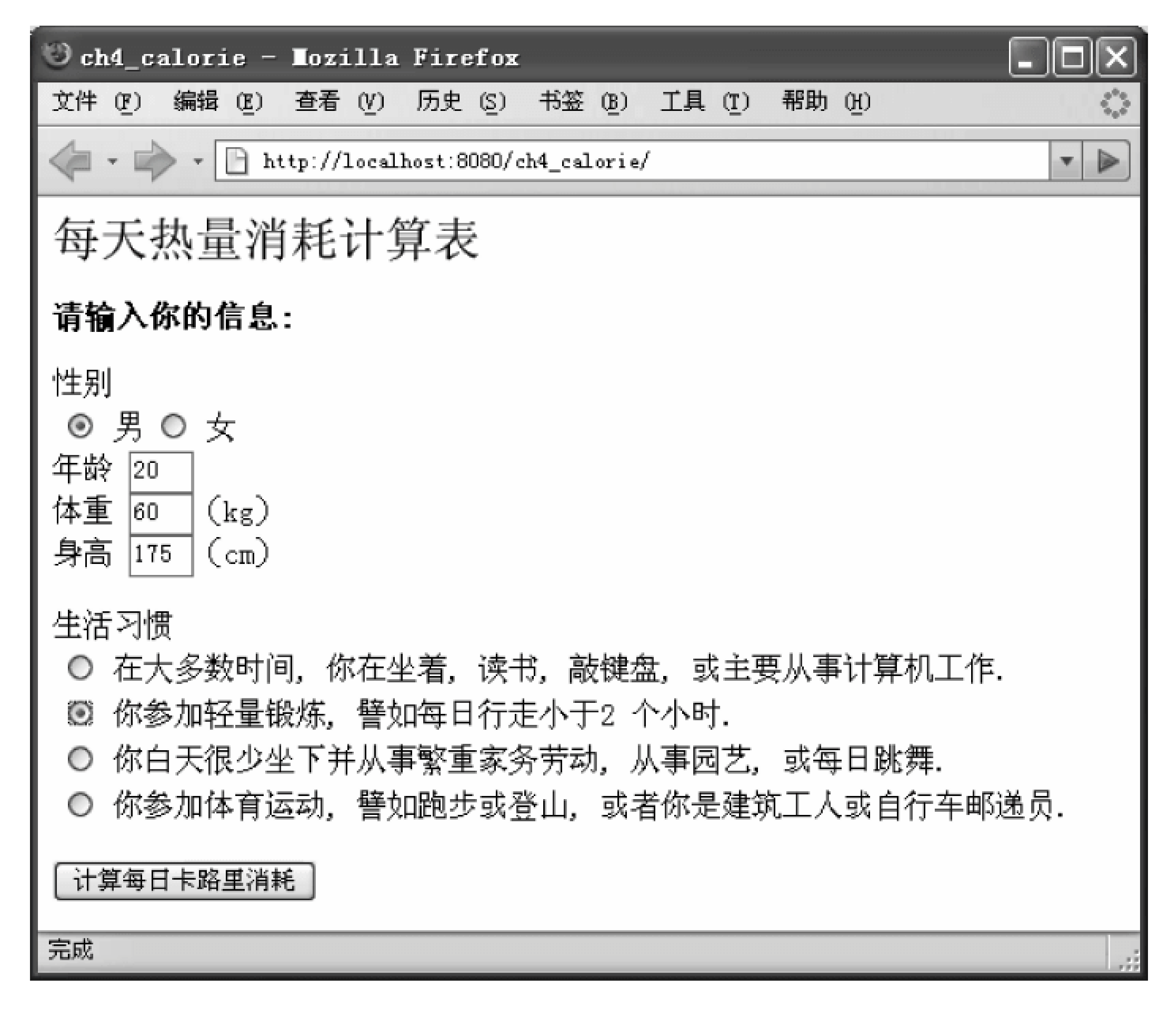


图 4-2　应用 ch4_calorie 欢迎页面

1. 受管 bean

Index.java 是该应用项目唯一的一个受管 bean，见代码清单 4-4。bean 中定义了若干属性：gender、age、height、weight 和 activity，分别用来接收用户在欢迎页面中输入的性别、年龄、身高、体重和生活习惯的值。这些属性都是可读写的，getter 方法会在欢迎页面显示

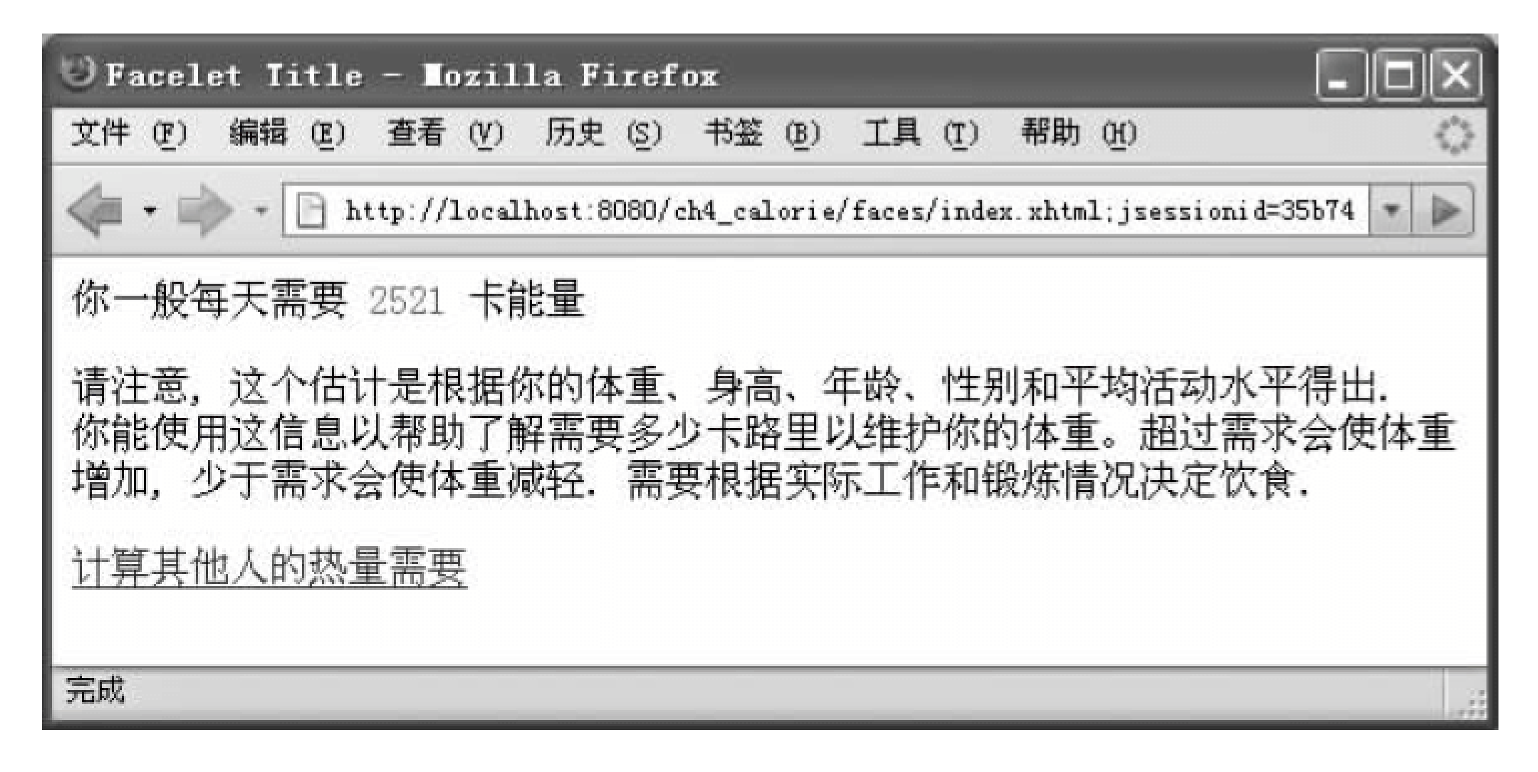


图 4-3 应用 ch4_calorie 响应页面

时调用，setter 方法会在页面提交时被调用。

bean 还定义了一个 List<SelectItem>型的只读属性 items，该属性在构造方法中初始化，为“生活习惯”单选按钮组提供各选项。

bean 中的 getResult 方法实现应用所需的计算功能，同时提供相应的只读属性 result。该属性在响应页面中被访问。

代码清单 4-4 受管 bean（Index.java）

```java
package bean;
import java.util.ArrayList;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.model.SelectItem;

@ManagedBean
@RequestScoped
public class Index {

  public Index(){
    items=new ArrayList<SelectItem>();
    SelectItem item=new SelectItem();
    item.setLabel("在大多数时间,你在坐着,读书,敲键盘,或主要从事计算机工作.");
    item.setValue(1.2);
    items.add(item);
    item=new SelectItem();
    item.setLabel("你参加轻量锻炼,譬如每日行走小于 2 个小时.");
    item.setValue(1.55);
    items.add(item);
    item=new SelectItem();
    item.setLabel("你白天很少坐下并从事繁重家务劳动,从事园艺,或每日跳舞.");
    item.setValue(1.725);
    items.add(item);
    item=new SelectItem();
```

```
  item.setLabel("你参加体育运动,譬如跑步或登山,或者你是建筑工人或自行车邮递员.");
  item.setValue(1.9);
  items.add(item);
}

private String gender="male";
public String getGender(){
  return gender;
}
public void setGender(String gender){
  this.gender=gender;
}

private int age;
public int getAge(){
  return age;
}
public void setAge(int age){
  this.age=age;
}

private int height;
public int getHeight(){
  return height;
}
public void setHeight(int height){
  this.height=height;
}

private int weight;
public int getWeight(){
  return weight;
}
public void setWeight(int weight){
  this.weight=weight;
}

private float activity=1.2f;
public float getActivity(){
  return activity;
}
public void setActivity(float activity){
  this.activity=activity;
}

```

```
    private List<SelectItem>items;
    public List<SelectItem>getItems(){
      return items;
    }

    public int getResult(){
      float fltBMR;
      //Calculate: Basal Metabolic Rate
      if (gender.equals("male")){
        fltBMR=(float)(66 +weight * 13.7 +height * 5 -age * 6.8);
      } else {
        fltBMR=(float)(655 +weight * 9.6 +height * 1.8 -age * 4.7);
      }
      //Get Total Energy Needs
      float fltTEN=(float)fltBMR * activity;
      return (int)fltTEN;
    }
}
```

2. JSF 页面

应用包含两个页面：欢迎页面 index. xhtml 和响应页面 response. xhtml。

index. xhtml(代码清单 4-5)页面主要提供一个表单，让用户输入为计算日能量消耗量所需的各种个人信息。

代码清单 4-5　index. xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>ch4_calorie</title>
  </h:head>
  <h:body>
    <h:outputText value="每天热量消耗计算表" style="font-size: 25px"/>
    <h:form>
      <p>
      <h:outputText value="请输入你的信息:" style="font-weight: bolder"/>
      </p>
      性别
      <h:selectOneRadio value="#{index.gender}" >
        <f:selectItem itemLabel="男" itemValue="male"/>
        <f:selectItem itemLabel="女" itemValue="female"/>
      </h:selectOneRadio>
      年龄
```

```
            <h:inputText value="#{index.age}" size="3"/><br/>
            体重
            <h:inputText value="#{index.weight}" size="3"/>(kg)<br/>
            身高
            <h:inputText value="#{index.height}" size="3"/>(cm)
            <p/>
            生活习惯
            <h:selectOneRadio value="#{index.activity}" layout="pageDirection">
              <f:selectItems value="#{index.items}"/>
            </h:selectOneRadio>
            <p/>
            <h:commandButton value="计算每日卡路里消耗" action="response"/>
        </h:form>
    </h:body>
</html>
```

response.xhtml(代码清单 4-6)页面通过访问 bean 的 result 属性,完成相应的计算功能并显示计算结果。

代码清单 4-6　response.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Facelet Title</title>
  </h:head>
  <h:body>
    你一般每天需要
    <h:outputText value="#{index.result}" style="color: red"/>
    卡能量
    <p></p>
    请注意,这个估计是根据你的体重、身高、年龄、性别和平均活动水平得出.<br/>
    你能使用这信息以帮助了解需要多少卡路里以维护你的体重。超过需求会使体重增加,
    少于需求会使体重减轻.需要根据实际工作和锻炼情况决定饮食.
    <p></p>
    <p><h:outputLink value="index.xhtml">计算其他人的热量需要</h:outputLink></p>
  </h:body>
</html>
```

4.9　多选类标记

多选类标记的相应组件类有共同的超类 UISelectMany,在该超类中定义了这些组件共同的组件族名 SelectMany。

4.9.1 标记功能

与单选标记一样,所有多选标记也都由子标记 f:selectitem 或 f:selectItems 指定选项。与单选组件不同,多选组件可以同时选择多个选项。多选组件的 value 属性值应该为数组或集合(Collection)。

1. h:selectManyCheckbox

复选框组,单击复选框图标或标签进行选择。

该标记在服务器端表示为一个 HtmlSelectManyCheckbox 组件实例。呈现时,由子标记指定的每个选项被呈现为一个 HTML input 元素,type 属性设置为"checkbox",name 属性为组件的客户端标识符,value 属性为选项值。每个 input 元素都有各自的 id。每个复选框选项的标签嵌套于 HTML label 元素中,其 for 属性指向对应的 input 元素。

2. h:selectManyListbox

多选列表框,size 属性指定显示的选项数。单击可选择一项,按住 Ctrl 键再单击可选择(或取消选择)其余各项,按住 Shift 键再单击可选择连续多项。

在服务器端表示为一个 HtmlSelectManyListbox 组件实例。组件呈现为 HTML select 元素,其 size 属性设置为组件的 size,元素包含 multiple="multiple"属性设置。由子标记指定的每个选项被呈现为一个 HTML option 子元素。

3. h:selectManyMenu

多选菜单,没有 size 属性,每次仅显示一个选项,可选择多项。

在服务器端表示为一个 HtmlSelectManyMenu 组件实例。组件呈现为 HTML select 元素,其 size 属性设置为 1,元素包含 multiple="multiple"属性设置。由子标记指定的每个选项被呈现为一个 HTML option 子元素。

实际上,目前浏览器并不支持多选菜单,h:selectManyMenu 标记呈现的效果仍然是多选列表,只不过一次仅显示一个选项。

在 Mazolla 浏览器中,多选菜单的显示效果甚至没有滚动按钮,只有一个选项标签。此时可以按住 Ctrl 键加箭头键移动各选项。当一个选项显示出来时,可以释放 Ctrl 键并按空格键选择或取消选择该选项。

4.9.2 常用属性

一些基本属性(如 id、value 等),以及单选类标记具有的一些属性也常用于多选类标记。除此之外,多选类标记还包括以下常用属性。

(1) collectionType:指定保存选择结果的集合的具体类型的完整类名。

多选组件的 value 属性值应该为数组或集合(Collection)。默认情况下,value 属性值是字符串数组。可以将多选标记的 value 属性绑定在受管 bean 的某个数组类型的属性上,如果数组元素类型不是字符串,那么 JSF 框架会调用标准转换器或自定义转换器对数据进行类型转换。

如果将标记的 value 属性绑定在受管 bean 的某个集合类型属性上,而该集合类型是接口或抽象类(如 List、Set 等),那么可以用该 collectionType 属性指定一个相应的具体类型,如 java. util. ArrayList。当提交表单时,JSF 框架会创建该属性指定的具体集合类型的实

例，并将选择结果保存在该 Collection 对象中，然后赋给受管 bean 的属性。与数组不同，保存选择结果的 Collection 对象的元素类型总是为字符串。

该属性适用于所有多选类标记。

（2）selectedClass、unselectedClass：指定 CSS 样式类，分别应用于已选和未选的选项的显示。这两个属性仅适用于 h:selectManyCheckbox 标记。

4.9.3 多选标记应用示例

该应用项目(ch4_selectmany)主要由一个页面和一个受管 bean 组成。页面上方是一个表单，包含一组复选框和一个动作按钮。当用户完成选择、单击动作按钮时，页面下方会显示用户选择的结果，如图 4-4 所示。



图 4-4 应用 ch4_selectmany 运行效果

1. 受管 bean

Index.java 是该应用唯一的受管 bean（代码清单 4-7）。bean 中定义了一个可读写的 result 属性，其类型为字符串数组，用于保存用户选择结果；一个只读的 msg 属性，是多选结果的一个字符串表示。

代码清单 4-7 受管 bean(Index.java)

```
package bean;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class Index {
  private String[] result;
  public String[] getResult(){
    return result;
  }
  public void setResult(String[] result){
    this.result=result;
  }
  public String getMsg(){
    if(result.length==0) return "没有选择。";
```

```
    StringBuffer sb=new StringBuffer("你的选择："+result[0]);
    for(int i=1;i<result.length;i++){
      sb.append("、").append(result[i]);
    }
    return sb.toString();
  }
}
```

2. JSF 页面

该应用仅包含一个 index.xhtml 页面(代码清单 4-8),其中复选框组标记的 value 属性绑定在受管 bean 的 result 属性上,输出文本标记的 value 属性绑定在受管 bean 的 msg 属性上。其中,输出文本组件仅在 result 属性不为 null 时才呈现,也就是说,只有在此时 getMsg 方法才会被调用。

代码清单 4-8　index.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>ch4_selectmany</title>
  </h:head>
  <h:body>
    <h:form>
      <h:outputText value="你掌握的编程语言："/>
      <h:selectManyCheckbox id="cs" value="#{index.result}" >
        <f:selectItem itemLabel="Java 语言" itemValue="Java"/>
        <f:selectItem itemLabel="C 语言" itemValue="C"/>
        <f:selectItem itemLabel="C++" itemValue="C++"/>
        <f:selectItem itemLabel="VisualBasic" itemValue="VisualBasic"/>
      </h:selectManyCheckbox>
      <p></p>
      <h:commandButton value=" OK "/>
    </h:form>
    <p></p>
    <h:outputText value="#{index.msg}" rendered="#{index.result!=null}"/>
  </h:body>
</html>
```

4.10　消息标记

JSF 在处理请求时可能会产生各种消息,包括:

- 标准转换器或自定义转换器在转换组件值时产生的错误消息;

• 设置了 required 属性的组件未获得输入值时产生的错误消息；
• 标准验证器或自定义验证器在验证组件值时产生的错误消息；
• 由应用代码根据需要自主产生的各种消息。

通常，这些消息应表示成 FacesMessage 对象，并放置在消息队列中，最后由 h:message 或 h:messages 消息组成呈现出来。

4.10.1 FacesMessage 类

每个消息包含三个属性：概要文本(summary)、详细文本(detail)和严重级别(severity)，用 FacesMessage 实例表示。FacesMessage 类的构造方法有：

```
• FacesMessage()                          //概要和详细文本均为 null,严重级别为信息级别
• FacesMessage(String summary)            //概要与详细文本相同,严重级别为信息级别
• FacesMessage(String summary,String detail)      //严重级别为信息级别
• FacesMessage(FacesMessage.Severity severity,String summary,String detail)
```

FacesMessage 类还定义了 4 个表示消息严重级别的类常量：

• FacesMessage.SEVERITY_INFO：信息级别。
• FacesMessage.SEVERITY_WARN：警告级别。
• FacesMessage.SEVERITY_ERROR：错误级别。
• FacesMessage.SEVERITY_FATAL：致命级别。

各种消息产生后首先被放入消息队列中，最后在"呈现响应"阶段被显示在页面上。每个消息可能属于某个特定组件(组件消息)，也可以属于整个页面(全局消息)。在把消息放入消息队列时，应该指明消息所属。

应用代码可以调用 FacesContext 对象的 addMessage 方法将消息放入队列：

```
FacesMessage msg1=new FacesMessage("msg1");
FacesMessage msg2=new FacesMessage("msg2");
//将消息 msg1 作为组件(客户端标识符为 f1:i1)消息放入消息队列。
FacesContext.getCurrentInstance().addMessage("f1:i1",msg1);
//将消息 msg2 作为全局消息放入消息队列。
FacesContext.getCurrentInstance().addMessage(null,msg2);
```

4.10.2 h:message 标记

h:message 标记称为消息标记，用于显示某个特定组件引发的消息。如果该组件引发了多个消息，该标记显示其中的第一个消息。

该标记在服务器端表示为一个 HtmlMessage 组件实例。呈现时，输出消息文本。如果包含 id、style、styleClass 等属性，消息文本嵌套于 HTML span 元素中。

如果要显示的是一个信息级别的消息，而标记指定了 infoStyle、infoClass 属性，则呈现时将用这些属性设置 HTML span 元素的 style、class 属性。对其他级别的消息，也会有类似的处理。

h:message 标记包含 id、style、styleClass、rendered、binding 等基本属性。但与其他标记不同，该标记没有 value 属性。除此之外，该标记还包含以下常用属性。

(1) for：指定消息源组件的标识符，即当前消息组件将显示该属性指定的组件引发的消息，String 型。该属性既适用于 h:message 标记，也适用于 h:outputLabel 标记（含义与作用有所不同），但不适用于 h:messages 标记。

(2) showSummary：指定是否显示消息的概要文本，boolean 型，默认值为 false。

(3) showDetail：指定是否显示消息的详细文本，boolean 型，默认值为 true。

(4) infoStyle：指定信息级别消息的 CSS 样式，String 型。

(5) infoClass：指定信息级别消息的 CSS 样式类，String 型。

(6) warnStyle：指定警告级别消息的 CSS 样式，String 型。

(7) warnClass：指定警告级别消息的 CSS 样式类，String 型。

(8) errorStyle：指定错误级别消息的 CSS 样式，String 型。

(9) errorClass：指定错误级别消息的 CSS 样式类，String 型。

(10) fatalStyle：指定致命级别消息的 CSS 样式，String 型。

(11) fatalClass：指定致命级别消息的 CSS 样式类，String 型。

(12) tooltip：指定在 showDetail 属性和 showSummary 属性均为 true 的情况下，是否以工具提示方式显示消息的概要文本，boolean 型，默认值为 false。

4.10.3 h:messages 标记

h:messages 标记称为消息组标记，可以显示所有全局消息和组件消息。如果一个组件引发了多个消息，这些消息都会被显示。

该标记在服务器端表示为一个 HtmlMessages 组件实例。组件呈现为 HTML table 元素或 HTML ul 元素，每个消息呈现为一个 HTML tr 元素（表格行）或 HTML li 元素（列表项）。组件的 id、style、styleClass 等属性将设置 table 元素或 ul 元素的相应属性（id、style 和 class 属性），组件的 infoStyle、infoClass 等属性将设置 tr 或 li 元素的相应属性（style 和 class 属性）。

h:messages 标记的常用属性与 h:message 标记类似，但存在着一些差异。

(1) h:messages 标记没有 for 属性。

(2) h:messages 标记特有的 globalOnly 属性，用于指定是否仅显示全局消息，默认值为 false。即在默认情况下，既显示全局消息，也显示组件消息。

(3) 与 h:message 标记不同，在消息组 h:messages 标记中，showSummary 属性的默认值是 true，showDetail 属性的默认值是 false。

(4) h:messages 标记包含 layout 属性，用于指定是以列表（list）形式还是表格（table）形式显示各消息，String 型，默认值为“list”。

h:panelGroup 标记也包含 layout 属性，但含义有所不同。

(5) h:messages 标记同样包含 infoStyle、infoClass、tooltip 等属性，但这些属性针对要显示的每一条消息。

4.11 论坛—登录与注册

本书将以一个简易论坛（取名为轻松论坛）的开发作为 JSF 的一个主要应用例子。这个应用系统的开发会随着相关知识点的引入、介绍而逐步展开。为保持相对的独立性，每引

入一些功能，都会创建一个 JSF 应用项目。

本节介绍论坛应用中的登录和注册功能，相应的应用项目名为 luntan_logandreg。与 2.4 节介绍的登录应用相比，该应用在功能上增加和增强了很多，主要包括：

(1) 增加了注册功能。除了用户名和密码，客户信息还包括性别、文化程度和电子邮箱等。

(2) 本应用包括三个页面：主页、登录页面和注册页面。主页能显示当前注册人数、在线人数。若用户还没有登录，能从主页导航至登录页面和注册页面。已登录的用户也可以退出(登离)。

(3) 登录页面和注册页面中包含消息组件。当登录失败或注册信息不合要求时，会在所在页面显示错误消息。当登录或注册成功时，将返回到主页。

该应用的运行效果如图 4-5 所示。其中：(a)和(b)分别是主页在用户未登录和用户已登录两种状态下的呈现结果；(c)是登录页面的呈现结果；(d)是注册页面的呈现结果。

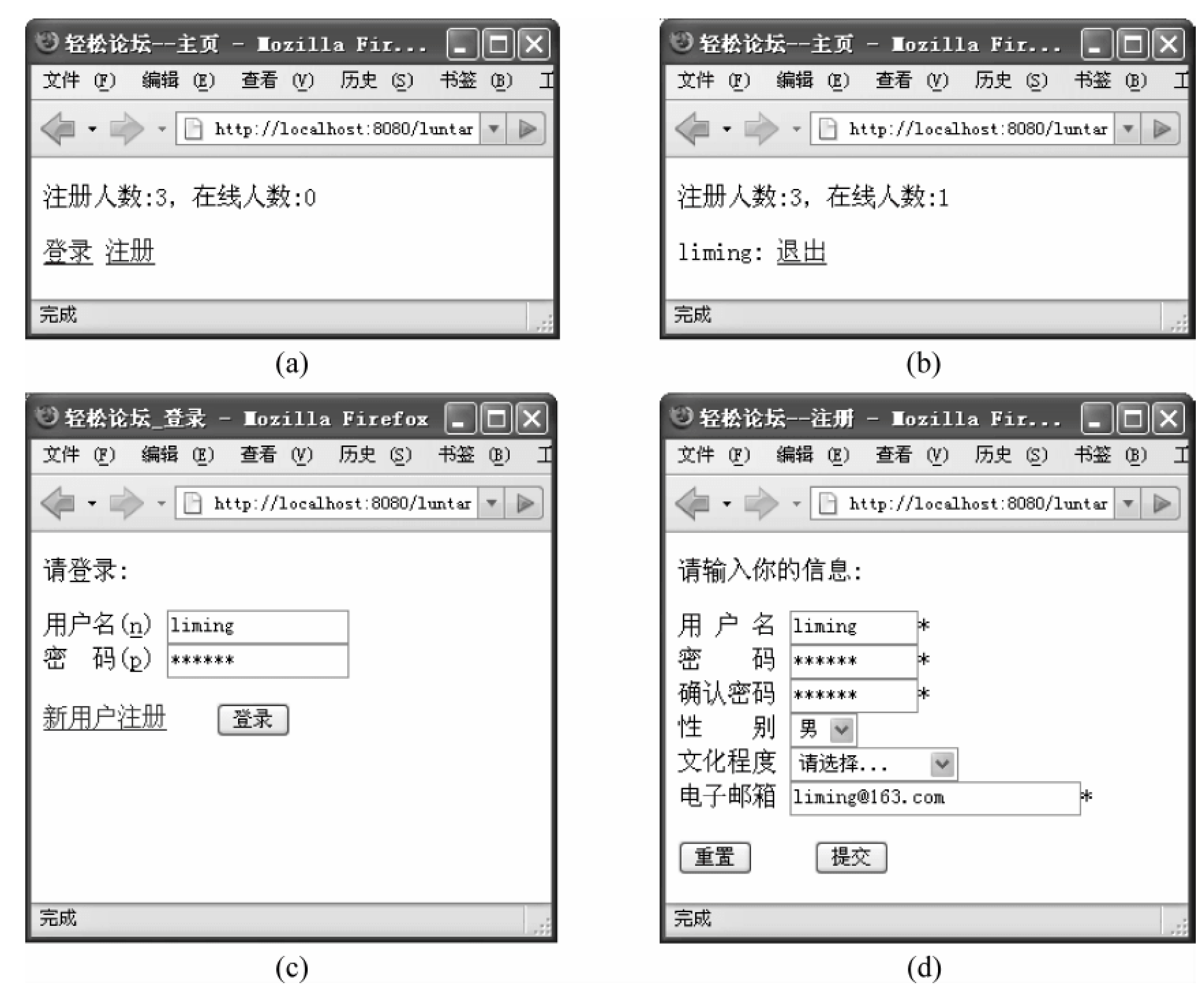


图 4-5　应用 luntan_logandreg 运行效果图

4.11.1　创建模型

首先在源包下创建一个名为 entity 的 Java 包，并在该包中创建一个表示用户的 Client 类。该类定义了 username、password 等 6 个实例变量，除了若干构造方法，每个实例变量都有相应的 getter 和 setter 方法。代码清单 4-9 给出了 Client 类的定义，其中各实例变量相应的 getter 和 setter 方法被省略了。

代码清单 4-9　Client.java

```
1. package entity;
```

```java

public class Client {
  private String username;
  private String password;
  private char gender;
  private String email;
  private Character degree;
  private char status;              //是否在线

  public Client(){
  }
  public Client(String username){
    this.username=username;
  }
  public Client(String username,String password,char gender,String email,char status){
    this.username=username;
    this.password=password;
    this.gender=gender;
    this.email=email;
    this.status=status;
  }

  //各实例变量相应的 getter 和 setter 方法

}
```

然后在源包下创建一个名 model 的 Java 包，并在该包中创建 DataBase 和 ClientManager 两个 Java 类。DataBase 类(代码清单 4-10)模拟应用的业务数据，即用户数据。这里，用一个 Map<String, Client>型对象存储注册用户的数据。ClientManager 类(代码清单 4-11)定义了应用所需的各种业务处理。

代码清单 4-10　DataBase.java

```java
package model;
import entity.Client;
import java.util.HashMap;
import java.util.Map;

public class DataBase {
  private static final Map<String,Client>clients=new HashMap<String,Client>();
  static {
    Client c1=new Client("loubuye","123456",'男',"loubuye@163.com",'0');
    c1.setDegree('1');
    clients.put("loubuye",c1);
    Client c2=new Client("zhaoyy","123456",'男',"zhaoyy@163.com",'0');
    clients.put("zhaoyy",c2);
```

```
    }
    public static Map<String,Client>getClients(){
      return clients;
    }
}
```

代码清单 4-11　ClientManager.java

```
package model;
import entity.Client;
import java.util.Collection;
import java.util.Map;

public class ClientManager {
    public Client findClientByName(String name){          //查找用户
      Client client=DataBase.getClients().get(name);
      return client;
    }
    public boolean insertClient(Client client){           //添加新用户
      String username=client.getUsername();
      boolean success=false;
      Map<String,Client>clients=DataBase.getClients();
      if(clients.get(username)==null){
        clients.put(username,client);
        success=true;
      }
      return success;
    }
    public int getNumOfClient(){                          //返回所有用户数
      return DataBase.getClients().size();
    }
    public int getNumOfOnline(){                          //返回状态为'1'的用户数
      Map<String,Client>map=DataBase.getClients();
      Collection<Client>coll=map.values();
      int n=0;
      for(Client c:coll){
        if(c.getStatus()=='1') n++;
      }
      return n;
    }
    public void logoff(Client client){                    //将用户状态设置为'0'
      client.setStatus('0');
    }
    public void login(Client client){                     //将用户状态设置为'1'
      client.setStatus('1');
    }
}
```

4.11.2 创建受管 bean

首先在源包下创建一个名为 util 的 Java 包,并在该包中创建 ELUtil 类。该类提供两个静态方法,可以分别计算一个 EL 表达式和获取一个受管 bean。这两个实用方法主要是为受管 bean 类的定义提供方便,比如一个受管 bean 要访问另一个受管 bean,可以先调用 ELUtil 类的 getBean 方法获得那个 bean。代码清单 4-12 给出了 ELUtil 类的代码。

代码清单 4-12　ELUtil.java

```
package util;
import javax.faces.application.Application;
import javax.faces.context.FacesContext;

public class ELUtil {
  //方法接收一个 EL 表达式字符串,计算并返回表达式的值
  public static Object evalEL(String el){
    FacesContext facesContext=FacesContext.getCurrentInstance();
    Application application=facesContext.getApplication();
    Object value=application.evaluateExpressionGet(facesContext,el,Object.class);
    return value;
  }
  //方法返回指定名称的受管 bean
  public static Object getBean(String name){
    return evalEL("#{"+name+"}");
  }
}
```

然后在源包下创建一个名为 bean 的 Java 包,并在该包中定义 Index、Login、Registry 和 SessionInfo 这 4 个受管 bean 类。其中,Index、Login 和 Registry 是请求作用域的受管 bean,SessionInfo 是会话作用域的受管 bean。

Index.java(代码清单 4-13)作为主页(index.xhtml)的支撑 bean,提供两个可读属性:numOfClient 和 numOfOnline,以及一个实现"退出"的动作方法。

代码清单 4-13　受管 bean(Index.java)

```
package bean;
import entity.Client;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import model.ClientManager;
import util.ELUtil;

@ManagedBean
@RequestScoped
public class Index {
  public int getNumOfClient(){                    //获得注册人数
    ClientManager cm=new ClientManager();
```

```
        return cm.getNumOfClient();
    }
    public int getNumOfOnline(){                    //获得在线人数
        ClientManager cm=new ClientManager();
        return cm.getNumOfOnline();
    }
    public String exit(){                           //用户“退出”动作方法
        SessionInfo sessinfo=(SessionInfo)ELUtil.getBean("sessinfo");
        Client client=sessinfo.getClient();
        sessinfo.setClient(null);
        ClientManager cm=new ClientManager();
        cm.logoff(client);
        return null;
    }
}
```

Login.java(代码清单 4-14)作为登录页面(login.xhtml)的支撑 bean,提供用户名和密码两个相对应的可读写属性,以及实现“登录”的动作方法。

代码清单 4-14　受管 bean(Login.java)

```
package bean;
import entity.Client;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.context.FacesContext;
import model.ClientManager;
import util.ELUtil;

@ManagedBean
@RequestScoped
public class Login {
    private String name;
    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name=name;
    }
    private String pw;
    public String getPw(){
        return pw;
    }
    public void setPw(String pw){
        this.pw=pw;
    }
```

```
    public String login(){                    //"登录"动作方法
      ClientManager cm=new ClientManager();
      Client client=cm.findClientByName(name);
      if(client==null){
        FacesMessage msg=new FacesMessage("用户名不存在");
        FacesContext.getCurrentInstance().addMessage("fi:username",msg);
        return null;
      } else if(client.getStatus()=='1'){
        FacesMessage msg=new FacesMessage("该用户已在线");
        FacesContext.getCurrentInstance().addMessage("fi:username",msg);
        return null;
      } else if(!client.getPassword().equals(pw)){
        FacesMessage msg=new FacesMessage("密码错误");
        FacesContext.getCurrentInstance().addMessage("fi:password",msg);
        return null;
      }
      SessionInfo sessinfo=(SessionInfo)ELUtil.getBean("sessinfo");
      cm.login(client);
      sessinfo.setClient(client);
      return "index";
    }
}
```

Registry.java(代码清单 4-15)作为注册页面(registry.xhtml)的支撑 bean,提供一个 Client 型的可读写属性 client、一个 String 型的可读写属性 password1,以及实现"注册"的动作方法。这里 client 属性必须事先实例化,password1 属性与确认密码相对应。

代码清单 4-15　受管 bean(Registry.java)

```
package bean;
import entity.Client;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.context.FacesContext;
import model.ClientManager;
import util.ELUtil;

@ManagedBean
@RequestScoped
public class Registry {
  private Client client=new Client();
  public Client getClient(){
    return client;
  }
  public void setClient(Client client){
    this.client=client;
```

```
  }
  private String password1;
  public String getPassword1(){
    return password1;
  }
  public void setPassword1(String password1){
    this.password1=password1;
  }
  public String registry(){                //注册“提交”动作方法
    if(!password1.equals(client.getPassword())){
      FacesMessage msg=new FacesMessage("两次输入的密码不一致");
      FacesContext.getCurrentInstance().addMessage("fi:password1",msg);
      return null;
    }
    ClientManager cm=new ClientManager();
    Client client1=cm.findClientByName(client.getUsername());
    if(client1!=null){
      FacesMessage msg=new FacesMessage("用户名已被使用");
      FacesContext.getCurrentInstance().addMessage("fi:username",msg);
      return null;
    }
    client.setStatus('1');
    boolean f=cm.insertClient(client);
    if(!f){
      FacesMessage msg=new FacesMessage("注册出错");
      FacesContext.getCurrentInstance().addMessage("fi:username",msg);
      return null;
    }
    SessionInfo sessinfo=(SessionInfo)ELUtil.getBean("sessinfo");
    sessinfo.setClient(client);
    return "index";
  }
}
```

SessionInfo.java(代码清单 4-16)是一个会话作用域的受管 bean,其 client 属性用于存储当前会话期已登录的用户对象。若 client 属性的值为 null,表示当前会话期用户没有登录。

代码清单 4-16　受管 bean(SessionInfo.java)

```
package bean;
import entity.Client;
import java.io.Serializable;
import javax.annotation.PreDestroy;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import model.ClientManager;

```

```
@ManagedBean(name="sessinfo")
@SessionScoped
public class SessionInfo implements Serializable {
  private Client client;
  public Client getClient(){
    return client;
  }
  public void setClient(Client client){
    this.client=client;
  }
  @PreDestroy
  public void shutdown(){
    if(client!=null){
      ClientManager cm=new ClientManager();
      cm.logoff(client);
    }
  }
}
```

4.11.3 创建 JSF 页面

该应用共有三个 JSF 页面：index. xhtml、login. xhtml 和 registry. xhtml。

主页 index. xhtml(代码清单 4-17)显示注册人数和在线人数。另外，如果当前会话期用户没有登录，页面会显示"登录"和"注册"超链接，单击它们可以分别导航至相应的页面，进行登录或注册。如果当前会话期用户已经登录，则显示用户名和"退出"超链接。

代码清单 4-17　主页(index. xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>轻松论坛--主页</title>
  </h:head>
  <h:body>
    <h:form>
      <p>
        <h:outputText value="注册人数:#{index.numOfClient},"/>
        <h:outputText value="在线人数:#{index.numOfOnline}"/>
      </p>
      <p>
        <h:commandLink value="登录" action="login"
                        rendered="#{sessinfo.client==null}"/>
        <h:outputText value=" "/>
```

```
        <h:commandLink value="注册" action="registry"
                      rendered="#{sessinfo.client==null}"/>
        <h:outputText value="#{sessinfo.client.username}:"
                      rendered="#{sessinfo.client!=null}"/>
        <h:commandLink value="退出" action="#{index.exit}"
                      rendered="#{sessinfo.client!=null}"/>
      </p>
    </h:form>
  </h:body>
</html>
```

登录页面 login.xhtml(代码清单 4-18)为用户提供登录界面。如果登录失败,会显示错误消息;如果登录成功,会自动导航至主页。另外,单击“新用户注册”超链接,可导航至注册页面。

代码清单 4-18　登录页面(login.xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>轻松论坛_登录</title>
  </h:head>
  <h:body>
    <p>
      <h:outputText value="请登录:"/>
    </p>
    <h:form id="fi">
      <h:outputLabel for="username" accesskey="n">
        用户名(<u>n</u>)
      </h:outputLabel>
      <h:inputText id="username" value="#{login.name}" size="15" maxlength="15"
          required="true" requiredMessage="用户名为必填项" styleClass="field1"/>
      <h:message for="username" styleClass="font4"/>
      <br/>
      <h:outputLabel for="password" accesskey="p">
        密　码(<u>p</u>)
      </h:outputLabel>
      <h:inputSecret id="password" value="#{login.pw}" size="15" maxlength="15"
          required="true" requiredMessage="密码为必填项" styleClass="field1"/>
      <h:message for="password" styleClass="font4"/>
      <p>
        <h:outputLink value="faces/registry.xhtml">新用户注册</h:outputLink>
          
        <h:commandButton id="submit" value="登录" action="#{login.login}"/>
```

```
        </p>
      </h:form>
    </h:body>
</html>
```

注册页面 registry. xhtml(代码清单 4-19)为用户提供注册界面。如果注册失败,会显示错误消息;如果注册成功,则该用户自动成为已登录用户,并导航至主页。

代码清单 4-19　registry. xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>轻松论坛--注册</title>
  </h:head>
  <h:body>
    <p>
      <h:outputText value="请输入你的信息:"/>
    </p>
    <h:form id="fi">
      用 户 名
      <h:inputText id="username" value="#{registry.client.username}"
          required="true" requiredMessage="用户名为必填项"
          size="10" maxlength="15"/> *
      <h:message for="username"/>
      <br/>
      密    码
      <h:inputSecret id="password" value="#{registry.client.password}"
          required="true" requiredMessage="密码为必填项"
          size="10" maxlength="15"/> *
      <h:message for="password"/>
      <br/>
      确认密码
      <h:inputSecret id="password1" value="#{registry.password1}"
          required="true" requiredMessage="密码为必填项"
          size="10" maxlength="15"/> *
      <h:message for="password1"/>
      <br/>
      性    别
      <h:selectOneMenu id="gender" value="#{registry.client.gender}">
        <f:selectItem itemLabel="男" itemValue="男"/>
        <f:selectItem itemLabel="女" itemValue="女"/>
      </h:selectOneMenu>
```

```
        <br/>
        文化程度
        <h:selectOneMenu id="degree" hideNoSelectionOption="true"
             value="#{registry.client.degree}">
          <f:selectItem itemLabel="请选择..." noSelectionOption="true"/>
          <f:selectItem itemLabel="高中或以下" itemValue="1"/>
          <f:selectItem itemLabel="大专" itemValue="2"/>
          <f:selectItem itemLabel="本科" itemValue="3"/>
          <f:selectItem itemLabel="硕士" itemValue="4"/>
          <f:selectItem itemLabel="博士或博士后" itemValue="5"/>
        </h:selectOneMenu>
        <br/>
        电子邮箱
        <h:inputText id="email" value="#{registry.client.email}"
           required="true" requiredMessage="email 为必填项"
           size="25" maxlength="25"/> *
        <h:message for="email"/>
        <p>
          <h:commandButton type="reset" value="重置"/>
           
           
          <h:commandButton value="提交" action="#{registry.registry}"/>
        </p>
      </h:form>
    </h:body>
</html>
```

4.12 小　　结

- 基于 Facelets 技术的 JSF 页面是一个 XHTML 页面，其文件扩展名为.xhtml。
- JSF 页面主要由 JSF 标记组成，也可包含普通的 HTML 标记、某些 JSTL 标记。
- JSF 标记包括 JSF HTML 标记、JSF 核心标记、JSF Facelets 标记、JSF 复合标记。
- JSF 核心标记通常执行一些与任何特定的呈现包无关的核心动作。
- 每个 JSF HTML 标记代表某种 UIComponent 组件与 JSF 框架的 HTML 呈现包中某个呈现器的一种组合。
- 绝大多数 JSF HTML 标记的标记名都由两部分组成，前半部分是相应组件类的组件族名，后半部分是相应呈现器的型名。
- JSF HTML 标记包括以下几种。

  基本输入类：h:inputText、h:inputSecret、h:inputTextArea、h:inputHidden。

  基本输出类：h:outputText、h:outputLabel、h:outputLink、h:outputFormat。

  图像：h:graphicImage。

  动作类：h:commandButton、h:commandLink。

二选一：h:selectBooleanCheckbox。

单选类：h:selectOneRadio、h:selectOneMenu、h:selectOneListbox。

多选类：h:selectManyCheckbox、h:selectManyMenu、h:selectManyListbox。

消息：h:message。

消息组：h:messages。

- 在 JSF 中，一个消息表示为一个 FacesMessage 实例。每个消息主要包括概要文本、详细文本和严重级别三个属性。

习　题　4

1. 简述 JSF HTML 标记与普通的 HTML 标记之间的关系。

2. 分别比较下面各组标记中各标记的主要功能。

(1) h:inputText、h:inputSecret

(2) h:outputText、h:outputLabel

(3) h:outputLink、h:commandLink

(4) h:commandButton、h:commandLink

(5) h:selectManyMenu、h:selectManyListbox

(6) h:message、h:messages

3. 解释下面各组属性的作用及其适用的标记。

(1) value、binding

(2) id、for

(3) style、styleClass

(4) title、label

(5) action、actionListener

(6) rendered、readonly、disabled

4. 在服务器端 Java 代码中，如何获取当前视图的某个组件实例？

5. 创建 JSF 应用 sh4_logandreg。该应用包括 3 个 JSF 页面、若干受管 bean 类和模型类，实现登录和注册功能。请按下列步骤完成应用开发。

(1) 在“源包”中分别创建 4 个 Java 包：bean、entity、model 和 util。

(2) 在 entity 包中创建一个表示客户的 Java 类 Client。下面是该类的不完整代码，请完善。

```
public class Client {
    private String username;            //用户名(每个用户的用户名是唯一的)
    private String password;            //密码
    private String realname;            //真实姓名
    private char sex;                   //性别
    private String phone;               //电话
    private String email;               //邮箱地址
    private String address;             //通信地址
```

```
    public Client(){}
    public Client(String username){
        //初始化实例变量
    }
    public Client(String username,String password,String realname,char sex,
        String email){
        //初始化实例变量
    }
    //为各实例变量提供相应的 getter 和 setter 方法
}
```

(3) 在 model 包中创建一个 DataBase 类,用于存放客户数据。下面是该类的不完整代码,请完善之。

```
public class DataBase {
    private static final Map<String,Client>clients=new HashMap<String,Client>();
    static {
        //初始化 clients: 添加 1 至若干个客户
    }
    public static Map<String,Client>getClients(){
        return clients;
    }
}
```

(4) 在 util 包中创建一个 Java 类 ELUtil,其中定义两个实用的静态方法。下面是该类的软件接口,请实现之(参见第 4.11 节例子)。

```
public class ELUtil {
    //方法接收一个 EL 表达式字符串,计算并返回表达式的值
    public static Object evalEL(String el);
    //方法接收一个受管 bean 的名称,返回该 bean 的引用
    public static Object getBean(String name);
}
```

(5) 在 model 包中创建一个实现客户管理业务的 ClientManager 类。下面是该类的软件接口,请实现之。

```
public class ClientManager {
    //根据用户名返回相应的客户。如果不存在相应的客户,返回 null
    public Client findClientByName(String name);
    //新添加一个客户,如果客户已存在、或其他原因无法完成添加,返回 false
    public boolean insertClient(Client client);
}
```

(6) 在 bean 包中创建一个受管 bean: 名称为 sessionBean1、类名为 bean.SessionBean1、作用域为会话作用域。bean 类中定义一个可读写的 Client 型属性 client。该属性表示当前会话期登入的客户,如果没有客户登录,该属性应为 null。

(7) 创建 JSF 页面 index. xhtml(欢迎页面或主页),以及相应的请求作用域、类名为 bean. Index 的受管 bean。页面的运行效果如图 4-6 所示。



(a) (b)

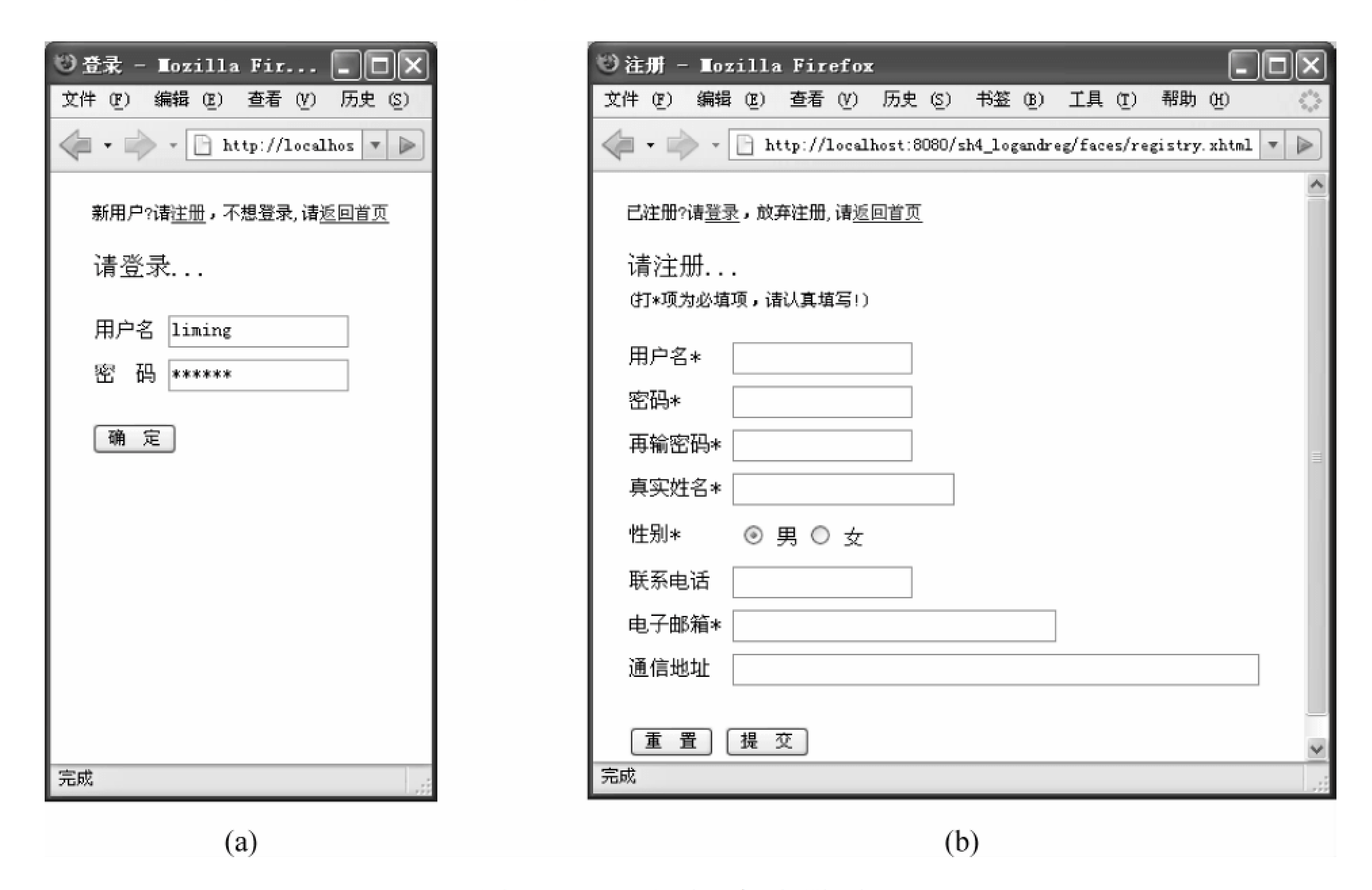
图 4-6　欢迎页面呈现效果

其中:

(a) 是客户未登录时的显示效果,单击"登录"超链接将转至登录页面 login. xhtml;单击"注册"超链接将转至注册页面 registry. xhtml。

(b) 是客户已登录时的显示效果,其中"李明"是登录客户的真实姓名。单击"退出"按钮将使客户退出登录状态,并重新显示该页面。

(8) 创建登录页面 login. xhtml,以及相应的请求作用域、类名为 bean. Login 的受管 bean。页面的运行效果如图 4-7(a)所示。



(a) (b)

图 4-7　登录页面与注册页面

单击"注册"超链接,将直接导航至注册页面 registry. xhtml;单击"返回首页"超链接,将直接导航至欢迎页面 index. xhtml。

用户单击"确定"按钮,应用将检查用户输入的用户名和密码是否正确。若用户名或密码不正确(如为空、用命名不存在、密码不正确等),返回本页面并显示错误信息。若用户名或密码正确,先获取相应的 Client 对象并保存到 sessionBean1 中的 client 属性中,然后导航至欢迎页面 index. xhtml 页面。

(9) 创建注册页面 registry. xhtml，以及相应的请求作用域、类名为 bean. Registry 的受管 bean。页面的运行效果如图 4-7(b)所示。

单击“登录”超链接，将直接导航至登录页面 login. xhtml；单击“返回首页”超链接，将直接导航至欢迎页面 index. xhtml。

单击“提交”命令按钮时，应用将检查用户输入的信息是否正确。若用户输入的信息不正确(如必填项为空)，返回本页面并显示相应的错误信息。若用户输入的信息正确，先创建相应的 Client 对象，并添加至 DataBase 类的 clients 属性中和保存到 sessionBean1 中的 client 属性中，然后导航至欢迎页面 index. xhtml。

注意：该项目不需要太关注页面的外观，只需完成相应的功能即可。可以在学完第 6 章后再来完善页面的布局和格式。

第 5 章　页面导航

本章主题：

- 导航处理及相关组件
- 隐式导航
- 导航规则及基于导航规则的导航
- 重定向
- h:link 与 h:button 标记
- 视图参数与可书签化 URL

对于一个 Web 应用来说，页面导航的设计好坏直接影响用户对应用的访问效率。从 JSF 1.2 到 JSF 2.0，JSF 提供的页面导航技术有了很多的扩充。从原先根据用户动作和业务逻辑确定目标视图的 POST 请求，到新的面向视图参数和目标视图的 GET 请求，本章将对它们做详细介绍。

5.1　导航概述

在 Web 应用中，所谓导航是指通过在页面上放置超链接或递交按钮，使访问者可以快速地从一个页面（视图）转至另一个页面（视图）。合适的导航能够方便访问者访问其所需的信息或请求相关的业务处理。

导航处理是指根据当前请求的上下文状态信息确定目标页面（视图）的过程。在 JSF 中，导航处理由导航处理器（NavigationHandler）完成，下面组件涉及导航处理：

- h:commandButton（动作按钮）；
- h:commandLink（动作超链接）；
- h:button（结果按钮）；
- h:link（结果超链接）。

对于动作类组件（h:commandButton 和 h:commandLink），导航处理发生在用户发出请求之后。当用户单击组件呈现的按钮或超链接提交请求时，将触发动作事件、调用动作方法，之后就由导航处理器确定要呈现的目标视图。这种导航处理也称为后置式导航处理。

对于结果类组件（h:button 和 h:link），导航处理发生在页面呈现前，即在组件呈现时，导航处理器就会确定目标视图。当用户单击组件呈现的按钮或超链接时，将直接转至之前已确定的目标视图。这种导航处理也称为前置式导航处理。

说明：h:outputLink 组件呈现的普通超链接是一种基本的导航技术，但其目标页面是在标记中直接指定的，所以并不需要 JSF 导航处理器进行导航处理。

无论是后置式导航处理还是前置式导航处理，其处理算法是一致的。导航处理器根据结果值、并结合其他上下文状态信息进行导航处理。下面是导航处理中会涉及的几个主要术语。

1. 动作值(action value)

动作组件的 action 属性值本身,可以是一个 EL 方法表达式,也可以是一个简单的字符串文字。

2. 结果值(outcome value)

是导航处理器进行导航处理的主要依据。结果值的可能情况包括:

- 动作组件的 action 属性指定的字符串文字。
- 动作组件的 action 属性指定的 EL 方法表达式的计算结果。
- 结果组件的 outcome 属性指定的字符串文字。
- 结果组件的 outcome 属性指定的值表达式的计算结果。

3. 视图 ID(view ID)

以"/"开头、相对于应用上下文路径的视图(页面)URL。

说明:

(1) 这里,action 是 h:commandButton 组件标记和 h:commandLink 组件标记的一个属性,而 outcome 是 h:button 组件标记和 h:link 组件标记的一个属性。

(2) 对于后置式导航处理,若 action 属性指定的是简单的字符串文字,则其动作值与结果值是相同的。

(3) 对于前置式导航处理,不牵涉动作值。

5.2 隐式导航

隐式导航是 JSF 2.0 新引入的特性,是指在开发人员没有指定相应的导航规则时,导航处理器所进行的导航处理。

当未定义导航规则或没有匹配的导航规则时,JSF 框架的导航处理器将把结果值看作是目标视图的视图 ID。如果结果值不以"/"开头,则在其前添加源视图相对于应用上下文路径的路径。如果结果值不包含文件扩展名,则在其后添加源视图的文件扩展名。

例如,源视图(/login.xhtml)的表单中包含如下的命令按钮标记:

```
<h:commandButton id="cm" value="OK" action="response"/>
```

当用户单击该按钮时,将会导航至视图 ID 为/response.xhtml 的目标视图。这里,虽然导航处理器参与了导航处理,但导航处理确定的目标页面视图总是某个固定的页面视图。这种导航称为静态导航。

又例如,源视图(/login.xhtml)的表单中包含如下的命令按钮标记:

```
<h:commandButton id="cm" value="OK" action="#{login.isUser}"/>
```

login 引用一个受管 bean,bean 类必须包含一个 isUser 方法。

```
public String isUser(){
    if(...){                          //验证成功
        return "success";
    } else {                          //验证失败
        return "failure";
```

```
    }
}
```

若验证成功，将会导航至视图 ID 为/success.xhtml 的目标视图。若验证失败，将会导航至视图 ID 为/failure.xhtml 的目标视图。这里，导航处理确定的目标页面视图不仅取决于用户单击了哪个按钮或超链接，也取决于当前请求的其他上下文状态信息，从而会导航至不同的目标页面视图。这种导航称为动态导航。

隐式导航的特点是不需要指定导航规则，目标视图 ID 由结果值直接指定。这种导航方式的优点是简单、直接，易于理解；缺点是不能对整个应用的导航模型进行集中管理，不便于维护。

隐式导航不仅可以实现静态导航，也可以实现动态导航；不仅适用于后置式导航，也适用于前置式导航。

5.3 基于导航规则的导航

当导航处理器进行导航处理时，首先寻找匹配的导航规则。如果存在匹配的导航规则，将依据导航规则确定目标页面视图。

5.3.1 导航规则

导航规则由 navigation-rule 元素及其子元素在/WEB-INF/faces-config.xml 或其他自定义的 Faces 配置文件中声明。下面是一个导航规则的声明示例。

```
<navigation-rule>
    <from-view-id>/index.jsp</from-view-id>
    <navigation-case>
        <from-action>#{login.isUser}</from-action>
        <from-outcome>ok</from-outcome>
        <if>#{login.valid}</if>
        <to-view-id>/success.xhtml</to-view-id>
    </navigation-case>
    …
</navigation-rule>
```

其中：

(1) 每一个 Faces 配置文件可以包含多个 navigation-rule 元素，每个 navigation-rule 元素声明一个导航规则。

(2) 每一个 navigation-rule 元素可以包含一个可选的 from-view-id 子元素，另外可以包含多个 navigation-case 子元素，每个 navigation-case 元素声明一个导航案例。

(3) 每一个 navigation-case 元素可以包含一个可选的 from-action 子元素、一个可选的 from-outcome 子元素、一个可选的 if 子元素，另外需要包含一个必选的 to-view-id 子元素。

下面是上述一些元素的含义及作用。

from-view-id：用于指定适用该导航规则的源页面视图的视图 ID。如果未指明该元素，

一个导航规则将适用于所有页面视图。一个页面视图的所有导航案例可以定义在一个导航规则里，也可以分散定义在多个导航规则里。

from-outcome：用于指定适用该导航案例的结果值。

from-action：用于指定动作值(即动作方法表达式本身)。源于同一个页面视图的多个请求，在调用各自的动作方法后可能会产生相同的结果值，此时可以通过动作值区分它们，以便匹配不同的导航案例。

if：该元素值应该是一个布尔型的值表达式，用于指定该导航案例是否匹配的动态条件。

to-view-id：用于指定目标页面视图的视图 ID。

5.3.2 导航算法

导航处理器以结果值为参数、导航规则和导航案例为依据，结合当前请求的其他上下文状态信息进行导航处理。下面是导航处理算法一个大致描述。

(1) 导航处理器首先会依次(在配置文件中从上到下)检查各导航规则，寻找相匹配的导航规则。相匹配的导航规则通常是指其 from-view-id 元素内容为源视图 ID 的导航规则。

(2) 当找到一个相匹配的导航规则时，导航处理器将依次检查其中的各导航案例，寻找相匹配的导航案例。相匹配的导航案例是指满足下面条件的导航案例：

- 如果包含 from-action 元素，其内容应等于动作值；
- 如果包含 from-outcome 元素，其内容应等于结果值；
- 如果包含 if 元素，其指定的值表达式的计算结果应为 true。

(3) 若发现一个相匹配的导航案例，其中不包含 if 元素，则：

- 若结果值为非 null，则其 to-view-id 元素的值即为目标视图 ID；
- 若结果值为 null，重新显示源视图。

(4) 若发现一个相匹配的导航案例，且其中包含 if 元素，则不管结果值是否为 null，其 to-view-id 元素的值即为目标视图 ID。

(5) 若未发现任何匹配的导航规则及导航案例，而结果值为 null，则重新显示源视图。

(6) 若未发现任何匹配的导航规则及导航案例，而结果值为非 null，则进行隐式导航。

说明：只有当结果值为 null 且不包含 if 元素时，当前视图作用域才会延续；否则在导航处理后，一个新的视图作用域将产生。

也可以在 WEB-INF 目录下创建其他的 Faces 配置文件(非 faces-config. xml)，然后在其中声明导航规则。这些配置文件的相对于应用上下文路径的路径列表(各路径间以逗号分隔)应作为名为 javax. faces. CONFIG_FILES 的上下文参数的初始值在 web. xml 文件中指定。下面是设置该上下文参数的一个示例。

```
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/faces-config_1.xml,/WEB-INF/faces-config_2.xml
    </param-value>
</context-param>
```

当存在多个 Faces 配置文件时，导航处理器按以下顺序检查各 Faces 配置文件中的导航规则和导航案例：

- 上下文参数 javax.faces.CONFIG_FILES 的值中指定的顺序。如对于上面示例，导航处理器将先检查 faces-config_1.xml 配置文件中的导航规则和导航案例，然后再检查 faces-config_2.xml 配置文件中的导航规则和导航案例。
- /WEB-INF/faces-config.xml 文件。该配置文件中的导航规则和导航案例会被最后检查，即使该配置文件出现在 javax.faces.CONFIG_FILES 上下文参数值中也是如此。

5.3.3 导航规则的进一步说明

这里对导航规则的一些特殊情况作进一步的说明。

1. 通配符

一个导航规则的 from-view-id 元素的内容可以以通配符 * 结尾。此时，判断该导航规则是否匹配的条件是指该元素内容的前缀(* 之前的内容)与源视图 ID 的前部是否相同。若存在多个这样的匹配导航规则，选择匹配前缀最长的一个。

假设源页面视图的视图 ID 是/subsys/group/page.xhtml，Faces 配置文件中含以下两个导航规则：

```
<navigation-rule>
    <from-view-id>/subsys/*</from-view-id>
    <navigation-case>
        ......
    </navigation-case>
</navigation-rule>
<navigation-rule>
    <from-view-id>/subsys/group/*</from-view-id>
    <navigation-case>
        ......
    </navigation-case>
</navigation-rule>
```

那么这两个导航规则都适用于上述源页面视图引发的导航，但后面一个导航规则更恰当，因为其匹配的前缀更长。

若一个导航规则的 from-view-id 元素的内容仅为一个 * ，或者根本不包含该元素，则该导航规则适用于任何源视图引起的导航处理。

在这里，一个导航规则可能适用于多个页面视图，反过来，一个源页面视图也可能存在多个相匹配的导航规则。对于一个具体的源页面视图，到底采用哪个导航规则，一要看哪个导航规则更恰当(视图 ID 匹配前缀较长)，二还要看该导航规则中是否有相匹配的导航案例。也就是说，如果最恰当的导航规则中没有相匹配的导航案例，那么将检查另一个相匹配的导航规则中是否有相匹配的导航案例。

2. 动态目标视图 ID

每个 navigation-case 元素必须包含一个 to-view-id 子元素。该子元素不仅可以直接指定目标视图 ID，也可以指定一个 EL 值表达式。后种情况下，具体的目标视图 ID 通过计算 EL 值表达式获得。例如：

```
<navigation-rule>
    <from-view-id>/main.xhtml</from-view-id>
    <navigation-case>
        <to-view-id>#{myBean.nextViewID}</to-view-id>
    </navigation-case>
</navigation-rule>
```

该导航规则适用于源自视图/main.xhtml 的导航。当导航处理时，名为 myBean 的受管 Bean 的 getNextViewID()方法被调用，以获取目标视图 ID。

5.4 重 定 向

当用户单击动作类按钮(h:commandButton)或超链接(h:commandLink)时会产生回送(postback)请求、引发相应组件的动作事件。在处理回送请求的“调用应用”阶段，动作方法被调用、并产生结果值，然后导航处理器以结果值为参数进行导航处理、确定目标页面视图，最后再由 JSF 框架将目标视图呈现给用户。这一过程的直接结果是：浏览器窗口中显示的是目标视图的内容，而浏览器的地址栏中仍显示源视图的 URL。在很多情况下，这种不一致是不适当的。利用重定向技术可以消除这种不一致。

重定向是由服务器和客户端浏览器共同完成的。首先，服务器并不直接将目标视图的内容呈现给客户端，而是先向客户端发送一个 HTTP 重定向响应。该响应仅包含状态行和响应头，其中状态码为 302，表示重定向；响应头包括 Location 域，指定重定向的目标视图的 URL。其次，浏览器接收到重定向响应后，会自动向目标页面发出请求，这一过程并不需要用户的介入。最终，浏览器地址栏中显示目标视图的 URL，浏览器窗口中显示目标视图的内容。

可以看出，重定向比原来的情况多了一个“请求—响应”过程，所以响应速度会变慢，但它可使浏览器上显示的页面内容与其地址栏中的 URL 保持一致。

当采用规则导航时，可以在 navigation-case 元素中插入 redirect 子元素，要求 JSF 框架重定向至目标视图。例如：

```
<navigation-rule>
    <from-view-id>/login.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>ok</from-outcome>
        <to-view-id>/index.xhtml</to-view-id>
        <redirect/>
    </navigation-case>
</navigation-rule>
```

如果采用隐式导航，可以在结果值中添加一个名为 faces-redirect、值为 true 的请求参数，以便实现重定向。下面是代码示例：

```
<h:commandButton value="确认" action="index?faces-redirect=true"/>
```

此时，问号(?)前面的 index 被取出以确定目标视图 ID，而 faces－redirect＝true 则指

示 JSF 框架重定向至目标视图。

5.5 h:link 与 h:button 标记

h:link 标记与 h: button 标记统称为结果类标记，其相应组件类有共同的超类 UIOutcomeTarget，在该超类中定义了两个组件共同的组件族名 OutcomeTarget。

结果类标记组件在呈现时，都会先根据 outcome 属性计算结果值，然后由导航处理器根据结果值确定目标视图。outcome 属性值可以是表示结果值的字符串，也可以是一个 EL 值表达式，结果值通过计算该值表达式获得。

与 h:commandButton 和 h:commandLink 标记不同，结果类标记 h:link 和 h:button 不需要置于 h:form 标记内，它们不会收集表单数据产生回送请求。结果类标记产生直接请求。

5.5.1 h:link

该标记在服务器端表示为一个 HtmlOutcomeTargetLink 组件实例。组件呈现为 HTML a 元素，其 href 属性值为为导航处理确定的目标视图的 URL。

单击由该标记呈现的超链接将产生一个对目标视图的直接请求。

标记组件呈现的超链接的文本既可以由该标记组件 value 属性指定，也可以由嵌套的文本或 h:outputText 标记指定，或者由嵌套的 h:graphicImage 标记指定超链接图像。

5.5.2 h:button

该标记在服务器端表示为一个 HtmlOutcomeTargetButton 组件实例，组件呈现为 HTML input 元素。默认情况下，input 元素的 type 属性值为“button”，即产生一个文本按钮，标记的 value 属性值指定按钮标题。若为标记指定了 image 属性，则 input 元素的 type 属性值为“image”，即产生一个图像按钮，input 元素的 src 属性值为标记的 image 属性指定的图像资源的 URL。无论哪种情况，input 元素都会包含一个 onclick 属性，其中的 JavaScript 代码可以产生超链接导航行为。

单击由该标记呈现的按钮将产生一个对目标视图的直接请求。

5.5.3 常用属性

除了 outcome、value 属性以及 id、render、binding、title、style、styleClass 等通用属性外，结果类标记还包含下面一些常用属性。

(1) fragment: 用于指定目标页面某个位置(锚点)的名称，使得当导航到目标页面时能直接定位于该属性指定的页面位置。

(2) disabled: 该属性适用于 h:link 标记，但不适用 h:button 标记，默认值为 false。若设置为 true，h:link 标记组件将呈现为 HTML span 元素，而不是 HTML a 元素。

(3) target: 该属性适用于 h:link 标记，但不适用 h:button 标记，用于指定目标资源打开的位置。默认情况下，目标资源会在当前窗口打开。

5.6　规则导航应用示例

本应用项目(ch5_navigation)包含一个 Faces 配置文件、一个受管 bean 类和三个 JSF 页面。应用的运行效果如图 5-1 所示。

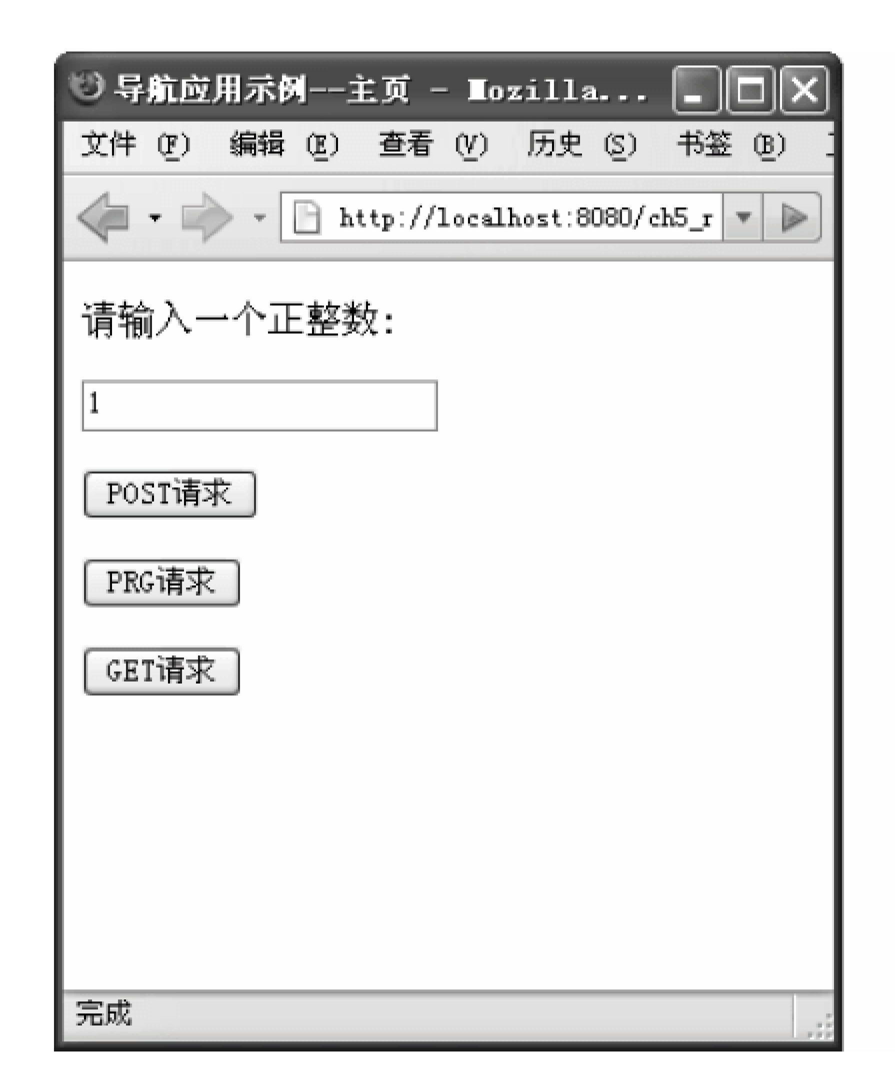






图 5-1　应用 ch5_navigation 运行效果

1. JSF 页面

该应用共有三个 JSF 页面。主页 index.xhtml(代码清单 5-1)主要包含一文本域和三个按钮。“POST 请求”和“PRG 请求”两个按钮都是由相应的 h:commandButton 标记产生的,“GET 请求”则是由 h:button 标记产生的。

代码清单 5-1　主页(index.xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>导航应用示例--主页</title>
  </h:head>
  <h:body>
    <h:form id="f">
      <p><h:outputLabel for="i" value="请输入一个正整数:"/></p>
      <p><h:inputText id="i" value="#{myBean.num}"/></p>
      <p><h:commandButton value="POST 请求"/></p>
      <p><h:commandButton value="PRG 请求" action="#{myBean.action}"/></p>
    </h:form>
    <p><h:button value="GET 请求"/></p>
```

```
  </h:body>
</html>
```

应用的另外两个页面分别是 response1. xhtml 和 response2. xhtml。这两个页面的功能都是显示受管 bean 中 num 属性的值,其中 response1. xhtml(代码清单 5-2)只是当 num 属性值为奇数时才被呈现,而 response2. xhtml(代码清单 5-3)则是当 num 属性值为偶数时才被呈现。之所以引入两个响应页面而非一个响应页面,完全是为了说明导航过程的需要。

代码清单 5-2　response1. xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>导航应用示例--响应 1</title>
  </h:head>
  <h:body>
    <p>这是一个奇数:#{myBean.num}</p>
    <h:link value="返回主页" outcome="index"/>
  </h:body>
</html>
```

代码清单 5-3　response2. xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>导航应用示例--响应 2</title>
  </h:head>
  <h:body>
    <p>这是一个偶数:#{myBean.num}</p>
    <h:link value="返回主页" outcome="index"/>
  </h:body>
</html>
```

两个响应页面中的“返回主页”超链接都由 h:link 标记产生,且都采用静态的隐式导航。下面主要讨论由主页到响应页面的导航过程。

2. 导航过程分析

由主页到响应页面的导航采用动态的规则导航。

首先给出应用中唯一的受管 bean 类的代码(代码清单 5-4),因为其中的一些方法和属性会在导航规则中被引用。其中,getValid()方法判断 num 属性值是否有效,即若属性值为正整数,返回 true,否则返回 false;getNextViewID()方法用于动态计算目标视图 ID。

代码清单 5-4　受管 bean(MyBean. java)

```java
package bean;

public class MyBean {

  private int num=1;
  public int getNum(){
    return num;
  }
  public void setNum(int num){
    this.num=num;
  }

  public String getNextViewID(){
    System.out.println(num);
    if(num%2==0){
      return "/response2.xhtml";
    } else {
      return "/response1.xhtml";
    }
  }
  public boolean getValid(){
    return num>=1;
  }
  public String action(){
    return null;
  }
}
```

然后给出应用的 Faces 配置文件内容(代码清单 5-5),其中包含受管 bean 的声明和导航规则的声明。

代码清单 5-5　faces-config. xml

```xml
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.0"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
  <managed-bean>
    <managed-bean-name>myBean</managed-bean-name>
    <managed-bean-class>bean.MyBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
  <navigation-rule>
    <from-view-id>/index.xhtml</from-view-id>
```

```
    <navigation-case>
      <from-action>#{myBean.action}</from-action>
      <if>#{myBean.valid}</if>
      <to-view-id>#{myBean.nextViewID}</to-view-id>
      <redirect/>
    </navigation-case>
    <navigation-case>
      <if>#{myBean.valid}</if>
      <to-view-id>#{myBean.nextViewID}</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

最后分析由主页到响应页面的导航过程。Faces 配置文件仅定义了一个导航规则，它适用于源自主页(index.xhtml)的导航。

该导航规则包含两个导航案例。在受管 bean 的 valid 属性值为 true 的情况下，前面那个导航案例适用于“PRG 请求”按钮的导航，后面那个导航案例使用于“POST 请求”和“GET 请求”按钮的导航。

这两个导航案例的次序不能颠倒，否则带 from-action 元素的导航案例将失效，三个按钮都将采用不带 from-action 元素的导航案例进行导航。

下面对三个按钮的导航进行逐一讨论。

当在文本域中输入一个整数、并按“POST 请求”按钮，将产生一个 POST 请求，输入的值被保存到受管 bean 的 num 属性中。在导航处理时，如果受管 bean 的 valid 属性值为 true(即 num 属性值为正整数)，将找到合适的导航案例，即后面那个导航案例。目标视图 ID 将通过计算 EL 值表达式 #{myBean.nextViewID} 获得，即若 num 属性值为奇数，则目标视图 ID 为/response1.xhtml，否则目标视图 ID 为/response2.xhtml。如果受管 bean 的 valid 属性值为 false(即 num 属性值不是正整数)，则找不到合适的导航案例，此时将重新显示主页。

当在文本域中输入一个整数、并按“PRG 请求”按钮时，总体情况与按“POST 请求”按钮时的一样，所不同的是：当找到合适的导航案例(前面那个)、并计算出目标视图 ID 后，并不是直接呈现目标视图，而是产生一个重定向(REDIRECT)响应，然后由客户端浏览器发出对目标视图的 GET 请求。

与上述两个按钮不同，“GET 请求”按钮的导航处理是在页面呈现前进行的。在导航处理时，如果受管 bean 的 valid 属性值为 true，将找到合适的导航案例(后面那个)。目标视图 ID 将通过计算 EL 值表达式 #{myBean.nextViewID} 获得。如果受管 bean 的 valid 属性值为 false，则找不到合适的导航案例，目标视图即为源视图。当页面呈现后，如果用户单击“GET 请求”按钮，则直接导航(GET 请求)至已经确定的目标页面。

5.7 视图参数与可书签化 URL

前面介绍，由 h:link 和 h:button 标记引发的导航是一种直接请求(GET 请求)。GET 请求通常用于查询某个资源(不改变服务器端资源的状态)、并通过视图显示该资源的状态，

所以被认为是等幂和安全的，其请求 URL 可以作为书签加以收藏。但在 Web 应用中，页面视图通常是动态的，即它可以显示某类资源中的任何一个资源的状态。所以在查询一个资源时，不仅需要指定相应的视图 ID，通常还需要指定与资源相关的一些查询参数。

本节介绍如何在视图中添加用于接收请求参数的组件——视图参数，如何在请求 URL 中设置查询参数，以及如何利用查询参数完成查询操作。

5.7.1 视图参数

视图参数标记 f:viewParam 在服务器端表示为 UIViewParameter 组件。该组件不会在客户端呈现，但可以接收从客户端发来的请求参数。

可以在 JSF 页顶部中添加视图参数标记 f:viewParam。视图参数作为当前视图的一种元数据，视图参数标记必须嵌套在 f:metadata 标记内。下面是 f:viewParam 标记的用法示例。

```
<f:metadata>
    <f:viewParam name="user" value="#{me.username}"/>
</f:metadata>
```

当请求被处理时，若存在名为 user 的查询参数，me 受管 bean 的 setUsername 方法将被调用并传入相应的查询参数。

一个 JSF 页面可以有任意数目的视图参数，或者说，在 f:metadata 标记内可以包含多个 f:viewParam 标记。

由于 UIViewParameter 类扩展 UIInput 类，所以第 4 章介绍的基本输入类标记的一些属性在视图参数标记中同样是适用的，比如可以为视图参数设置转换器、验证器等。与回送请求一样，对包含视图参数的视图的直接请求，JSF 框架在处理时，也要经历恢复视图、应用请求值、处理验证、更新模型值、调用应用和呈现响应各个阶段。

5.7.2 设置请求参数

无论是 h:button 和 h:link 触发的 GET 请求，还是 h:commandButton 和 h:commandLink 引起的重定向 GET 请求，如果目标视图包含视图参数，那么通常应该设置相应的请求参数。下面介绍设置请求参数的各种渠道。

1. 基于视图参数设置请求参数

在导航处理确定目标视图后，可以基于目标视图的视图参数现有值自动设置相应的请求参数（请求参数名与视图参数名相同）。这样当导航至目标视图时，这些请求参数将送至目标视图并重新设置其中的视图参数。

有多种方法导致导航处理器利用视图参数设置请求参数，这些方法有各自的适用范围。

方法 1：在 h:link 或 h:button 标记中，将 includeViewParams 属性设置为 true。下面是这种方法的示例。

```
<h:link outcome="index" includeVieweParams="true" value="OK"/>
```

这种方法不仅适用于隐式导航，也适用于规则导航。

方法 2：在 h:commandButton 和 h:commandLink 引起的重定向 GET 请求中，可以在结果

值中添加一个名为 includeViewParams、值为 true 的请求参数。下面是这种方法的示例。

```
<h:commandButton action="index?faces-redirect=true&amp;includeViewParams=true"
    value="确认"/>
```

这种方法仅适用于隐式导航，当然结果值既可以在 action 属性中直接指定，也可以通过动作方法返回。

方法 3：在导航规则声明中，将 redirect 元素的 include-view-params 属性值设置为 true。下面是这种方法的示例。

```
<navigation-case>
    <from-outcome>ok</from-outcome>
    <to-view-id>/index.xhtml</to-view-id>
    <redirect include-view-params=true/>
</navigation-case>
```

这种方法适用于规则导航，既适用于 h:commandButon 和 h:coomandLink 标记，也适用于 h:button 和 h:link 标记。

2. 在结果值中指定请求参数

如果采用隐式导航，可以在结果值中指定请求参数。

下面代码演示了 h:link 标记(或 h:button 标记)在结果值中指定请求参数的方法。

```
<h:link outcome="index?user=liming" value="OK"/>
```

结果值既可以在 outcome 属性中直接指定，也可以通过计算某 EL 值表达式获得。如果指定多个请求参数，各参数之间用"&"分隔(在 JSF 页面中，应使用其特殊表示法，即：&)。

下面代码演示了 h:commandLink 标记(或 h:commandButton 标记)在结果值中指定请求参数的方法。

```
<h:commandButton action="index?faces-redirect=true&amp;user=liming" value=
    "确认"/>
```

结果值既可以在 action 属性中直接指定，也可以通过动作方法返回。

3. 在导航规则中指定请求参数

如果采用规则导航，可以在导航规则声明中，用嵌套于 redirect 元素的 view-param 子元素指定请求参数。

```
<redirect>
    <view-param>
        <name>user</name>
        <value>liming</value>
    </view-param>
</redirect>
```

其中，redirect 元素中可以包含多个 view-param 子元素，每个 view-param 元素指定一个请求参数。

这种设置请求参数的渠道既适用于 h:commandButon 和 h:coomandLink 标记，也适用于 h:button 和 h:link 标记。

4. 利用 f:param 子标记设置请求参数

可以在 h:link 或 h:button 标记内嵌入 f:param 子标记，以便为 GET 请求设置请求参数。

```
<h:link outcome="index" value="OK">
    <f:param name="user" value="liming"/>
</h:link>
```

上面标记呈现后产生的结果类似于如下：

```
<a href="/jsfapp/faces/index.xhtml?user=liming">OK</a>
```

说明：也可以在 h:commandButon 和 h:coomandLink 标记内嵌入 f:param 子标记，但由此指定的请求参数只是作为回送请求（POST 请求）的请求参数，而不会作为之后引发的重定向请求的请求参数。

上面介绍了四种为 GET 请求设置请求参数的渠道。可以同时采用多种渠道设置请求参数，但上述第 2 种和第 3 种渠道不会同时出现。另外，这些渠道有一定的优先级，由低到高依次为：

- 基于视图参数设置请求参数。
- 在结果值中指定请求参数或在导航规则中指定请求参数。
- 利用 f:param 子标记设置请求参数。

若不同的渠道设置了同名的请求参数，则按上述顺序，后面的代替前面的。

5.7.3 preRenderView 系统事件

正如前面所述，当通过 GET 方式请求一个页面视图时，其中的请求参数将被用于设置目标视图的视图参数。但与 h:commandButton 和 h:commandLink 标记引起的 POST 请求能够触发动作事件不同，由 h:button 和 h:link 标记引起的 GET 请求并不会触发动作事件，因此也无法指定动作方法、进而在“调用应用”阶段执行该方法。那么如何能够在合适的时机利用视图参数来检索和定位视图实际要呈现的资源呢？

preRenderView 是一种组件系统事件（ComponentSystemEvent）。这种事件在呈现响应阶段、在视图实际呈现前触发，事件源是当前要呈现的视图的根（UIViewRoot）。一般情况下，可以在视图根上为该种事件注册一个监听方法，然后在监听方法中实现对要呈现的资源的检索和定位。

JSF 核心标记 f:event 能够在某组件上为某种感兴趣的组件系统事件注册一个监听方法。下面代码在视图根上注册了一个监听方法，用于监听 preRenderView 事件。

```
<f:metadata>
    ……
    <f:event type="preRenderView" listener="#{me.method}"/>
</f:metadata>
```

要在视图根上注册监听方法，一般应将 f:event 标记嵌套于 f:metadata 标记内。type 属性指定感兴趣的事件类型，可以采用事件类型的短名字（shortName），如 preRenderView，也可以

使用事件类型的完整类名，如 javax.faces.event.PreRenderViewEvent。listener 属性指定一个 EL 方法表达式，作为事件的监听方法。组件系统事件的监听方法的方法头一般如下：

```
public void method(ComponentSystenEvent cse)
```

如果监听方法不需要了解事件的有关信息，也可以不包含形参。在方法体中，如果处理事件时出现异常情况，不想再继续处理该事件，可以抛出一个 AbortProcessingException 型例外。

5.8 论坛—发表主题与回复

本节继续 4.11 节介绍的应用项目(luntan_logandreg)，对其进行功能扩充并做必要的修改。为保持原先项目的独立性，这里新创建一个名为 luntan_input 的 JSF 应用项目，并从原先项目复制所有的 JSF 页面、源包中的所有 Java 包和 Java 类。

与原先的应用项目 luntan_logandreg 相比，新的应用项目 luntan_input 主要增加了以下功能：统计和显示主题数、新建主题、统计和显示某一主题的回复数、对某个主题进行回复等。

图 5-2 显示了该应用的运行效果。只有登录的用户才能新建主题或对主题进行回复。图 5-2(a)是主页，是用户登录后的显示效果。单击主页中的“新建主题”按钮将进入“新建主



图 5-2 应用 luntan_input 运行效果图

题”页面，如图 5-2(b)所示。在“新建主题”页面，输入主题的标题和内容后单击“提交”按钮会返回主页。图 5-2(c)是“查看回复”页面，单击其中的“回复主题”按钮将进入“回复主题”页面，如图 5-2(d)所示。在“回复主题”页面，输入回复内容后单击“回复”按钮会返回“查看回复”页面。

在该应用中，“查看回复”页面与主页之间没有建立导航路径。要显示“查看回复”页面，需要在浏览器地址栏输入其地址，并提供一个名称为 tid 的请求参数，用以指定主题编号，如：

```
http://localhost:8080/luntan_input/faces/showReply.xhtml?tid=1
```

5.8.1 扩充模型

在该应用中，需要添加主题和回复等论坛应用的核心业务数据，并提供相应的业务处理方法。

首先在 entity 包中定义分别表示主题帖和回复帖的 Topic 类和 Reply 类。

Topic 类定义了 id、title、content 等 9 个实例变量，除了若干构造方法，每个实例变量都有相应的 getter 和 setter 方法。Reply 类还实现了 Comparable 接口，其中的 compareTo 方法可以按最后回复时间(lastreplytime)比较两个主题的大小，最后回复时间越大，相应的主题帖越小。

代码清单 5-6 给出了该类的定义，其中各实例变量相应的 getter 和 setter 方法被省略了。

代码清单 5-6 Topic.java

```
package entity;
import java.util.Date;

public class Topic implements Comparable<Topic>{
  private Integer id;                              //主题帖编号
  private String title;                            //标题
  private String content;                          //内容
  private Date createtime;                         //创建时间
  private int clickcount;                          //点击数
  private int replycount;                          //回复数
  private Date lastreplytime;                      //最后回复时间
  private Client client;                           //创建人
  private Client client1;                          //最后回复人

  public Topic(){
  }
  public Topic(Integer id){
    this.id=id;
  }
  public Topic(Integer id,String title,String content,Date createtime,
               int clickcount,int replycount,Date lastreplytime){
    this.id=id;
```

```
    this.title=title;
    this.content=content;
    this.createtime=createtime;
    this.clickcount=clickcount;
    this.replycount=replycount;
    this.lastreplytime=lastreplytime;
  }

  //各实例变量对应的 getter 方法和 setter 方法

  @Override
  public int compareTo(Topic t){
    return -(lastreplytime.compareTo(t.lastreplytime));
  }
}
```

Reply 类定义了 id、content、replytime 等 5 个实例变量，除了若干构造方法，每个实例变量都有相应的 getter 方法和 setter 方法。Reply 类还实现了 Comparable 接口，其中的 compareTo 方法可以按回复时间（replytime）比较两个回复的大小，回复时间越大，相应的回复帖也越大。

代码清单 5-7 给出了该类的定义，其中各实例变量相应的 getter 和 setter 方法被省略了。

代码清单 5-7　Reply.java

```
package entity;
import java.util.Date;

public class Reply implements Comparable<Reply>{
  private Integer id;                          //回复帖编号
  private String content;                      //内容
  private Date replytime;                      //回复时间
  private Topic topic;                         //相应的主题
  private Client client;                       //回复人

  public Reply(){
  }
  public Reply(Integer id){
    this.id=id;
  }
  public Reply(Integer id,String content,Date replytime){
    this.id=id;
    this.content=content;
    this.replytime=replytime;
  }

```

```
    //各实例变量对应的 getter 方法和 setter 方法

    @Override
    public int compareTo(Reply r){
      return replytime.compareTo(r.replytime);
    }

}
```

然后在 model 包的 DataBase.java 类中定义现有的主题帖和回复帖数据。主题帖保存在一个 List<Topic>型的表中，回复帖保存在一个 List<Reply>型的表中。当类装入初始化时，自动创建 10 个主题帖，但没有创建任何回复帖。

代码清单 5-8 列出了该类的代码，其中原有的涉及用户数据的代码被忽略了。

代码清单 5-8　DataBase.java(部分)

```
public class DataBase {
  ......
  private static final List<Topic>topics=new ArrayList<Topic>();
  private static final List<Reply>replys=new ArrayList<Reply>();

  static {
    ......
    String s="_uuuuuuuuuu_";
    Date date;
    Topic t=null;
    for(int i=1;i<11;i++){
      date=new Date();
      t=new Topic(i,"title"+s+i,"content"+s+i,date,0,0,date);
      t.setClient(c1);
      t.setClient1(c1);
      topics.add(t);
    }
  }
  ......
  public static List<Topic> getTopics(){
    return topics;
  }
  public static List<Reply> getReplys(){
    return replys;
  }
}
```

最后提供相关的业务处理代码。这里新建 TopicManager 和 ReplyManager 两个 Java 类，两个类都属于 model 包。

TopicManager 类(代码清单 5-9)定义了涉及主题的业务方法，包括往主题表添加一个

主题、为某主题添加一个回复等。getTopics 方法返回所有主题,且各主题按最后回复时间降序排序。

代码清单 5-9　TopicManager.java

```
package model;
import entity.Reply;
import entity.Topic;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class TopicManager {
  public int insertTopic(Topic topic){                //添加一个主题
    List<Topic>topics=DataBase.getTopics();
    int id=0;
    synchronized(topics){
      id=topics.size()+1;
      topic.setId(id);
      topics.add(topic);
    }
    return id;
  }
  public Topic getTopicById(int id){                  //根据 id 返回相应的主题
    List<Topic>topics=DataBase.getTopics();
    Topic t=null;
    for(Topic t1:topics){
      if(t1.getId()==id){
        t=t1;
        break;
      }
    }
    return t;
  }
  public List<Topic>getTopics(){                      //返回所有的主题
    ArrayList<Topic> topics=new ArrayList<Topic>(DataBase.getTopics());
    Collections.sort(topics);
    return topics;
  }
  public void replyTopic(Topic topic,Reply reply){    //为主题添加一个回复
    topic.setReplycount(topic.getReplycount()+1);
    topic.setClient1(reply.getClient());
    topic.setLastreplytime(reply.getReplytime());
  }
}
```

ReplyManager 类(代码清单 5-10)定义了涉及回复的业务方法,包括往回复表添加一个

回复、返回指定主题的所有回复等。getReplys(int)方法返回指定主题的所有回复，且各回复按回复时间升序排序。

代码清单 5-10　ReplyManager.java

```
package model;
import entity.Reply;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ReplyManager {
  public int insertReply(Reply reply){     //添加一个回复
    List<Reply>replys=DataBase.getReplys();
    int id=0;
    synchronized(replys){
      id=replys.size()+1;
      reply.setId(id);
      replys.add(reply);
    }
    return id;
  }
  public List<Reply>getReplys(int id){    //根据主题 id 返回该主题所有的回复
    List<Reply> replys=DataBase.getReplys();
    List<Reply> list=new ArrayList<Reply>();
    for(Reply r:replys){
      if(r.getTopic().getId()==id) list.add(r);
    }
    Collections.sort(list);
    return list;
  }
}
```

5.8.2　创建“新建主题”页

“新建主题”页显示一个表单，用户可以输入新主题的标题和内容。当用户单击“提交”按钮时，应用系统将创建一个主题并添加到主题表中，然后重定向至主页。

作为支撑该页面的受管 bean，InputTopic 类定义了与主题的标题和内容相对应的可读写的 title 和 content 属性。另外为主题的内容提供了一个验证方法，确保内容长度不会超出 1000 个字符。代码清单 5-11 是该 bean 类的代码，其中各属性的 getter 方法和 setter 方法被忽略了。

代码清单 5-11　InputTopic.java

```
package bean;
import entity.Client;
import entity.Topic;
```

```
import java.util.Date;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;
import model.TopicManager;
import util.ELUtil;

@ManagedBean
@RequestScoped
public class InputTopic {
  private String title;                                   //主题标题
  private String content;                                 //主题内容

  //各属性对应的 getter 方法和 setter 方法

  public void validate(FacesContext context,UIComponent comp,Object val)
                      throws ValidatorException{    //主题内容验证方法
    String str=(String)val;
    if(str.length()>1000){
      throw new ValidatorException(new FacesMessage(FacesMessage.SEVERITY_INFO,
                                   null,null));
    }
  }
  public String create(){                                  //"提交"主题动作方法
    SessionInfo sessinfo=(SessionInfo)ELUtil.getBean("sessinfo");
    Client client=sessinfo.getClient();
    if(client!=null){
      Topic topic=new Topic();
      Date date=new Date();
      topic.setTitle(title);
      topic.setContent(content);
      topic.setClient(client);
      topic.setCreatetime(date);
      topic.setClickcount(0);
      topic.setReplycount(0);
      topic.setClient1(client);
      topic.setLastreplytime(date);
      TopicManager tm=new TopicManager();
      tm.insertTopic(topic);
    }
    return "index?faces-redirect=true";
  }
}
```

代码清单 5-12 列出了“新建主题”页面(inputTopic. xhtml)的内容。当用户单击“提交”按钮时会调用受管 bean 的 create 方法,完成添加新主题的功能。

代码清单 5-12　inputTopic. xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html">
 <h:head>
   <title>轻松论坛_新建主题</title>
 </h:head>
 <h:body>
   <h:form id="f1">
     <p>
       <h:outputText value="新建主题:"/>
     </p>
     <p>
       <h:outputLabel for="input1" value="标题:"/>
       <h:inputText id="input1" value="#{inputTopic.title}"
            maxlength="50" size="50"
            required="true" requiredMessage="标题不能为空"/>
       <h:message for="input1" styleClass="font4"/>
     </p>
     <p>
       <h:outputLabel for="input2" value="内容:"/>
       <h:message for="input2" styleClass="font4"/>
       <br/>
       <h:inputTextarea id="input2" value="#{inputTopic.content}"
          rows="5" cols="50"
          required="true" requiredMessage="内容不能为空"
          validator="#{inputTopic.validate}"
          validatorMessage="长度不能超过 1000 字符"/>
     </p>
     <p>
       <h:commandButton value=" 提 交 " action="#{inputTopic.create}"/>
     </p>
   </h:form>
 </h:body>
</html>
```

5.8.3　修改主页

主页(index. xhtml)原先的主要功能是显示注册人数和在线人数。现在需要增加一些功能,包括显示现有的主题数,以及提供一个“新建主题”按钮,单击该按钮可以导航至“新建

主题”页面。

为此，需要对主页及相关的受管 bean 作出相应的修改。受管 bean(Index. java)需要添加一个类型为 List<Topic>的可读的 topic 属性，代码清单 5-13 给出了相关的代码(该 bean 类原先的代码被忽略了)。

代码清单 5-13　Index. java(部分)

```
public class Index {
  private List<Topic> topics;            //主题表
  public Index(){
    TopicManager tm=new TopicManager();
    topics=tm.getTopics();
  }
  public List<Topic> getTopics(){
    return topics;
  }
  ......
}
```

对主页(index. xhtml)的修改主要是在原有内容的后面添加了一个表单，见代码清单 5-14，其中该页面原先的内容被忽略了。

代码清单 5-14　index. xhtml(部分)

```
<h:body>
  ......
  <h:form>
    <p>
      <h:outputText value="主题数:#{index.topics.size()}"/>
    </p>
    <p>
      <h:commandButton value="新建主题" action="inputTopic?faces-redirect=true"
                           disabled="#{sessinfo.client==null}"/>
    </p>
  </h:form>
</h:body>
```

当单击“新建主题”按钮时，将采用隐式导航直接重定向至“新建主题”页面。如果用户还没有登录，该按钮是不可用的。

5.8.4　创建“回复主题”页面

访问“回复主题”页面时需要提供一个名为 tid 的请求参数。该请求参数将作为视图参数，用以指定某个主题编号。“回复主题”页面允许对指定的主题进行回复。

作为支撑该页面的受管 bean，InputReply 类定义了与回复主题相关的属性，包括 tid、topic 和 content。其中，tid 与视图参数绑定在一起，topic 是根据 tid 获得的主题(页面上要显示主题的标题)，content 接收回复的内容。另外，验证方法 validate 用于确保回复内容的

长度不会超过 1000 个字符，动作方法 reply 创建一个回复，然后将回复添加至回复表并修改相关主题的属性。代码清单 5-15 是该 bean 类的代码。

代码清单 5-15　InputReply.java

```
package bean;
import entity.Client;
import entity.Reply;
import entity.Topic;
import java.io.Serializable;
import java.util.Date;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;
import model.ReplyManager;
import model.TopicManager;
import util.ELUtil;

@ManagedBean
@ViewScoped
public class InputReply implements Serializable {
  private int tid;                          //主题 id
  public int getTid(){
    return tid;
  }
  public void setTid(int tid){
    this.tid=tid;
    TopicManager tm=new TopicManager();
    topic=tm.getTopicById(tid);
  }
  private Topic topic;                      //根据主题 id 获得的当前主题
  public Topic getTopic(){
    return topic;
  }
  private String content;                   //回复内容
  public String getContent(){
    return content;
  }
  public void setContent(String content){
    this.content=content;
  }
  public void validate(FacesContext context,UIComponent comp,Object val)
             throws ValidatorException{     //回复内容验证方法
```

```
    String str=(String)val;
    if(str.length()>1000){
      throw new ValidatorException(new FacesMessage(FacesMessage.SEVERITY_INFO,
                                null,null));
    }
  }
  public String reply(){                    //“回复”动作方法
    SessionInfo sessinfo=(SessionInfo)ELUtil.getBean("sessinfo");
    Client client=sessinfo.getClient();
    if(client!=null){
      Reply reply=new Reply();
      reply.setContent(content);
      reply.setClient(client);
      reply.setTopic(topic);
      reply.setReplytime(new Date());

      ReplyManager rm=new ReplyManager();
      rm.insertReply(reply);

      TopicManager tm=new TopicManager();
      tm.replyTopic(topic,reply);
    }
    return "showReply?faces-redirect=true&amp;tid="+tid;
  }
}
```

“回复主题”页面 inputReply.xhtml 见代码清单 5-16。当用户单击“回复”按钮时会调用受管 bean 中的 reply 方法。方法在完成相应的保存任务后，将采用隐式导航导航至“查看回复”页面。

代码清单 5-16　inputReply.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <f:metadata>
    <f:viewParam name="tid" value="#{inputReply.tid}"/>
  </f:metadata>
  <h:head>
    <title>轻松论坛_回复主题</title>
  </h:head>
  <h:body>
    <h:form id="f1">
      <p>
```

```
        <h:outputText value="回复主题: "/>
      </p>
      <p>
        <h:outputText value="标题: "/>
        <h:outputText value="#{inputReply.topic.title}"/>
      </p>
      <p>
        <h:outputLabel for="input2" value="内容: "/>
        <h:message for="input2"/>
        <br/>
        <h:inputTextarea id="input2" value="#{inputReply.content}"
            rows="5" cols="50"
            required="true" requiredMessage="内容不能为空"
            validator="#{inputTopic.validate}"
            validatorMessage="长度不能超过 1000 字符"/>
      </p>
      <p>
        <h:commandButton value=" 回 复 " action="#{inputReply.reply}"/>
      </p>
    </h:form>
  </h:body>
</html>
```

需要特别注意的是，这里的 InputReply 受管 bean 是视图作用域的。下面分析一下若把其设置成请求作用域会产生什么问题。当用户请求“回复主题”页面时，必须提供 tid 请求参数。该请求参数被作为视图参数、并用于设置受管 bean 的 tid 属性。在设置 tid 属性时，会获取 topic 属性值。若受管 bean 是请求作用域，则视图呈现后，该受管 bean 将消失，原有的 tid 和 topic 将不再存在。这样，当用户单击“回复”按钮后，那么在服务器端就只有回复的内容，而没有相关主题的数据，所以接下来的业务处理将无法进行。

当然，也可以通过改写“回复”按钮的标记，比如用嵌套的 f:param 标记将当前视图参数设置为请求参数，使得单击“回复”按钮产生的请求仍然包含原先的 tid 请求参数。就像原先的 GET 请求该页面一样，这次的 POST 请求该页面同样会用请求参数去设置受管 bean 的 tid 和 topic 属性。但这一设置过程发生于 JSF 请求处理生命周期的“更新模型值”阶段。如果在“处理验证”阶段，发现回复内容不合法（如空或长度大于 1000 个字符），那么将会跳过其他阶段、直接进入“呈现响应”阶段。这样，原来的 tid 将不复存在。

将 InputReply 受管 bean 设置为视图作用域可以避免以上问题的出现。

5.8.5 创建“查看回复”页面

“查看回复”页面显示某一主题的标题、回复数，并提供一个“回复主题”按钮，单击该按钮可携带一个请求参数 tid（当前主题的编号）导航至“回复主题”页面。

作为支撑该页面的受管 bean，ShowReply 类定义了与主题及其回复相关的属性，包括 tid、topic 和 replys。其中，主题编号 tid 与视图参数绑定在一起，topic 是根据 tid 获得的主题（页面上要显示主题的标题），replys 是一个包含当前主题所有回复的表。代码清单 5-17

是该 bean 类的代码。

代码清单 5-17　ShowReply.java

```
package bean;
import entity.Reply;
import entity.Topic;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import model.ReplyManager;
import model.TopicManager;

@ManagedBean
@RequestScoped
public class ShowReply {
  private int tid;                          //主题 id
  public int getTid(){
    return tid;
  }
  public void setTid(int tid){
    this.tid=tid;
  }
  public void initView(){
    TopicManager tm=new TopicManager();
    topic=tm.getTopicById(tid);
    ReplyManager rm=new ReplyManager();
    replys=rm.getReplys(tid);
    Reply reply=new Reply();
    reply.setContent(topic.getContent());
    reply.setClient(topic.getClient());
    reply.setId(0);
    reply.setReplytime(topic.getCreatetime());
    reply.setTopic(topic);
    replys.add(0,reply);
  }
  private Topic topic;                      //当前主题
  public Topic getTopic(){
    return topic;
  }
  private List<Reply> replys;               //回复表
  public List<Reply> getReplys(){
    return replys;
  }
  public String goReply(){                  //"回复主题"动作方法
    return "inputReply?faces-redirect=true&amp;tid="+tid;
```

```
    }
}
```

“查看回复”页面 showReply.xhtml 见代码清单 5-18。页面在开始处设置了一个视图参数 tid,并指定了一个 preRenderView 事件的监听方法,即受管 bean 的 initView 方法。当用户请求该页面时必须提供一个表示某主题编号的 tid 请求参数,该请求参数将设置视图参数 tid,进而设置受管 bean 的 tid 属性。在页面呈现前将引发 preRenderView 事件,受管 bean 的 initView 方法被调用。initView 方法首先根据 tid 获取主题 topic,以及该主题的所有回复的表 replys,然后基于当前主题创建一个回复插入到表 replys 的最前端。其中,方法的后面一部分处理对于本应用并不需要,主要是为后续的扩充做准备的。

代码清单 5-18　showReply.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <f:metadata>
    <f:viewParam name="tid" value="#{showReply.tid}"/>
    <f:event type="preRenderView" listener="#{showReply.initView}"/>
  </f:metadata>
  <h:head>
    <title>轻松论坛--查看回复</title>
  </h:head>
  <h:body>
    <h:form>
      <p>
        <h:outputText value="回复数:#{showReply.replys.size()-1}"/>
      </p>
      <p>
      <h:commandButton value="回复主题" action="#{showReply.goReply}"
                          disabled="#{sessinfo.client==null}"/>
      </p>
    </h:form>
    <h:outputText value="主题:#{showReply.topic.title}"/>
  </h:body>
</html>
```

至此,该应用的所有内容都已经介绍。可以看出,相关的业务数据已比较充分和完善,但页面显示的信息还比较贫乏。比如,主题表已被获取并存放在受管 bean 中,但主页中只显示出主题数,并没有显示各主题的数据;某个主题的回复表已经生成,但“查看回复”页面仅显示出回复数,并没有显示各回复的信息。在第 6 章介绍完数据表格组件后,论坛应用将主要关注这方面的扩充。

5.9 小　　结

- 导航处理是指根据当前请求的上下文状态信息确定目标页面(视图)的过程。涉及导航处理的组件包括以下几个。
  h:commandButton(动作按钮)。
  h:commandLink(动作超链接)。
  h:button(结果按钮)。
  h:link(结果超链接)。
- 导航处理由导航处理器完成。导航处理器在进行导航处理时,首先寻找导航规则,如果存在导航规则,则按导航规则进行导航;否则按默认约定进行导航。前者称为规则导航,后者称为隐式导航。
- 重定向是指不直接把目标视图呈现给客户端,而是先发送一个重定向响应,客户端接收到重定向响应后,再自动向目标视图发出请求。
- 动作类标记产生 POST 请求,采用后置式导航;结果类标记产生 GET 请求,采用前置式导航。
- 动作类标记产生 POST 请求时,会收集表单数据作为请求参数;结果类标记产生 GET 请求时,可以通过多种渠道设置请求参数。
- 视图参数标记 f:viewParam 在服务器端表示为 UIViewParameter 组件。该组件不会在客户端呈现,但可以接收从客户端发来的请求参数。

习　题　5

1. 术语解释。
- 动作值;
- 结果值;
- 视图 ID。

2. 简述重定向导航的过程,什么时候需要重定向?重定向有什么好处?
3. 试分别比较隐式导航与规则导航、后置式导航与前置式导航各自的特点。
4. 如何为 h:button 和 h:link 触发的 GET 请求设置请求参数?
5. 修改应用项目 sh4_logandreg(第 4 章习题 5)中各页面的导航方式,具体要求如下:

(1) 对主页 index. xhtml 的“登录”和“注册”超链接,采用 h:link 标记、GET 请求登录页面 login. xhtml 和注册页面 registry. xhtml。

(2) 对主页 index. xhtml 的“退出”超链接,采用 h:commandLink 标记产生 POST 请求,处理完相应的业务功能后,再重定向至主页本身。

(3) 对登录页面 login. xhtml 的“注册”和“返回首页”超链接,采用 h:link 标记、GET 请求注册页面 registry. xhtml 和主页 index. xhtml。

(4) 对登录页面 login. xhtml 的“确定”按钮,采用 h:commandButton 标记产生 POST 请求,然后处理相应的业务功能:如果登录成功,重定向至主页;如果登录失败,返回登录页

面本身。

（5）对注册页面 registry. xhtml 的“登录”和“返回首页”超链接，采用 h:link 标记、GET 请求登录页面 login. xhtml 和主页 index. xhtml。

（6）对注册页面 registry. xhtml 的“提交”按钮，采用 h:commandButton 标记产生 POST 请求，然后处理相应的业务功能：如果注册成功，重定向至主页；如果注册失败，返回注册页面本身。

第 6 章　页面布局与数据表格

本章主题：

- CSS 技术
- 面板标记
- 数据表格标记
- 编辑数据表格
- 分页显示技术

本章涉及页面布局与数据表格两个话题。好的页面布局与设计对一个 Web 应用来说占有举足轻重的地位。CSS 是一种用于格式化页面的标准技术，可以实现页面布局和设置内容显示格式等功能。本章首先介绍 CSS 的定义和应用技术，但有关 CSS 属性的内容，读者可参阅其他图书，本书不做专门介绍。

在 Web 应用中，经常会出现要显示成批数据的情况，这时需要用到数据表格组件。表格组件既可用于显示数据集，也可辅助 CSS 完成页面布局。本章后面几节将介绍表格等相关组件及其应用技术。

6.1　CSS 技术

CSS(Cascading Style Sheets，层叠样式表)是一种用于指定网页呈现格式的计算机语言。使用 CSS 技术的优点有：

(1) 网页的呈现内容与呈现格式分离，可以提高网页代码的可读性；

(2) 对网站所有或部分网页的呈现格式进行统一的定义，可以确保网站具有一致的呈现风格；

(3) 一个呈现格式信息通常会被一个网页或多个网页反复使用，因此可以减少页面的代码数量，提供下载速度；

(4) 网站所有或部分网页的呈现格式信息集中定义，便于修改和维护，可以降低网站的开发和维护工作量。

6.1.1　定义 CSS

一个样式表由若干样式组成，每个样式指定一组表示呈现格式的属性。定义样式的一般格式如下：

```
<选择符>{<样式属性名>:<属性值>;...}
```

其中，选择符指定该样式作用的对象。样式属性名与属性值之间用冒号(:)分隔，各属性名一值对之间用分号(;)分隔，最后一个分号可以省略。下面介绍一些常用的选择符。

1. 标记选择符

格式：

```
<标记名>{/*样式属性设置*/}
```

可以用网页的某种标记的名称作为选择符，称为标记选择符。由此定义的样式将作用于所有由选择符指定的标记的呈现。

假设有如下样式定义：

```
h3{color:blue; font-family:黑体}
```

那么网页中所有<h3>…</h3>之间的文字(3 级标题)用蓝色、黑体显示。

2. 类选择符

格式：

```
.<类名>{/*样式属性设置*/}   或   <标记名>.<类名>{/*样式属性设置*/}
```

选择符可以由点号(.)紧跟一个自定义的类名表示，称为类选择符。由此定义的样式将作用于所有 class 属性值为指定类名的标记的呈现。

类选择符前面可以包括标记名，此时的样式仅作用于 class 属性值为指定类名的特定标记的呈现。

假设有如下样式定义：

```
.c1{font-size:12px;letter-spacing:2px}
p.c2{width:90%;background-color:blue}
```

那么网页中所有 class 属性值为 c1 的标记，其内容将用字体大小 12 像素、字间距 2 像素显示；class 属性值为 c2 的 p 标记，呈现时的段落宽度为容器宽度的 90％、背景颜色为蓝色。

3. ID 选择符

格式：

```
#<id值>{/*样式属性设置*/}
```

ID 选择符由♯号紧跟一个自定义的 id 值组成。由此定义的样式将作用于 id 属性值为指定 id 值的标记的呈现。

假设有如下样式定义：

```
#em{font-weight:blod}
```

那么 id 属性值为 em 的标记的内容将以粗体显示。

4. 属性选择符

格式：

```
[<标记属性名>=<属性值>]   或   <标记名>[<标记属性名>=<属性值>]
```

属性选择符指定网页标记的某个属性及属性值，前面也可以包含一个标记名。由此定义的样式将作用于包含指定属性及属性值的所有标记或特定标记的呈现。

假设有如下样式定义：

```
input[type=password]{background-color:gray}
```

那么网页中所有口令域将以灰色背景显示。

5. 伪类

格式：

```
:<伪类名>   或   <选择符>:<伪类名>
```

伪类可以为指定标记的特定状态设置样式属性。标记由选择符指定，状态由伪类名指定。常用的伪类名有：

- link：未被访问过的状态，只能用于带 href 属性的 a 标记。
- visited：已被访问过的状态，只能用于带 href 属性的 a 标记。
- hover：鼠标划过时的状态。
- active：鼠标点击按下时的状态。
- focus：得到焦点时的状态。

假设有如下样式定义：

```
a:link {color:steelblue;text-decoration:none}
a:visited {color:steelblue;text-decoration:none}
a:hover {color:red;text-decoration:underline}
input[type=text]:focus {background-color:whitesmoke}
```

那么页面中未被访问过和已被访问过的超链接都以钢青色显示且没有下划线；鼠标划过的超链接将以红色显示且有下划线；文本域组件聚焦时会以白雾色作为背景颜色。

说明：一般的浏览器都支持关于超链接的伪类，但对其他标记的伪类并不一定支持。

6.1.2 使用 CSS

上面主要介绍如何定义样式，本节将介绍样式表的几种存在方式以及如何将其应用于页面及其元素。

1. 定义内部样式表

内部样式表定义于页面内，其样式可应用于页面内的标记。内部样式表在 style 标记内定义，而 style 标记一般放置在页面的 head 标记内。下面代码演示了定义内部样式表的一般格式。

```
<h:head>
    <style type="text/css">
    <!--
        p {font-size:12px;color:steelblue}
        .cone {font-family:楷体_gb2312;text-align:center}
    -->
    </style>
</h:head>
```

这里，把整个样式表包含在 HTML 注释内，可以避免不支持 CSS 的浏览器把样式表内容直

接显示出来。而对于支持 CSS 的浏览器,则会对其进行分析,并将其中的样式应用于页面中的相关标记。由于 CSS 已成为一种标准,一般的浏览器都应该支持 CSS,所以通常也可以省略其中的 HTML 注释标记。

2. 定义内联样式

这里,内联样式是指在页面标记的 style 属性中指定的样式。内联样式仅作用于所在的标记。下面代码演示了定义内联样式的格式。

```
<h:outputLabel for="i1" value="姓名: " style="font-size: 12px;font-family: 黑体"/>
<h:inputText id="i1" size="10" style="font-size:12px;background-color: whitesmoke"/>
```

内联样式用法简单,其应用效果也很容易观察,但违背了 CSS 技术的初衷,即没有将要显示的内容和内容的显示格式分离,所以内联样式一般只在页面调试时使用。另外,内联样式也可用于那些需要特殊样式的标记,为这些标记指定特定的样式属性,或用它覆盖普通样式中的某些样式属性。

3. 链接外部样式表

内联样式和内部样式表都不能很好地体现 CSS 技术的优势。一个定义好的样式,不能仅仅用于一个标记或一个页面,而是应该能用于所有需要它的页面和标记。

可以将一个样式表定义保存在一个单独的文件中,称为样式表文件。样式表文件是一个文本文件,其扩展名应该为.css。

在页面中可以用 link 标记链接一个样式表文件。link 标记用于在当前页面与样式表文件(或其他外部文档)之间建立连接,使得页面中的标记可以使用被链接的样式表中的样式。与定义内部样式表一样,link 标记一般也应该写在页面的 head 标记内。下面代码演示了链接外部样式表的方法。

```
</h:head>
    <link rel="stylesheet" type="text/css" href="css/cssone.css"/>
</h:head>
```

其中,rel 属性指定当前页面文档与被链接外部文档之间的关系,在链接样式表文件时,通常取值为“stylesheet”;type 属性指定被链接外部文档的 MIME 类型,对于样式表文件,其值总是“text/css”;href 属性用于指定外部样式表文件的地址。在上面代码中,被链接的样式表文件 cssone.css 应该存放在当前页面所在目录的子目录 css 中。

一个样式表文件可以被多个页面文档链接。反过来,一个页面也可以链接多个样式表文件。一个页面除了可以链接外部样式表,还可以同时用 style 标记定义内部样式表。但需要注意,这些样式表的链入或定义的先后次序会影响其中的样式的优先级。

说明:可以将 rel 属性指定为“alternate stylesheet”,并通过 title 属性指定一个标题。这样,被链接的外部样式表就成为一个可替换的样式表。页面的读者可以通过指定标题来选择要应用于页面的样式。Firefox、Opera 等浏览器支持这种交互功能。

4. 导入外部样式表

导入外部样式表是指用@import 符号在一个样式表中链接另一个样式表。也就是说一个样式表除了能定义自己的样式外,还可以包含另一个样式表的样式,但@import 符号必须出现在其他样式定义之前。下面代码演示了导入外部样式表的方法。

```
<h:head>
    <style type="text/css">
    <!--
        @import "css/cssone.css";
        @import "css/csstwo.css";
        p {font-size:12px;color:steelblue}
    -->
    </style>
</h:head>
```

这个代码在内部样式表中导入 url 地址分别为“css/cssone. css”和“css/csstwo. css”的两个样式表。除了可用于内部样式表，@import 符号也可用于外部样式表，即在一个外部样式表中导入另外的外部样式表。

说明：link 是一个 HTML 标记，用于在页面中链接一个外部样式表或其他类型的外部文档。@import 是一个 CSS 符号，用于在一个样式表中导入另外的外部样式表。

5. h:outputStylesheet 标记

这是一个 JSF HTML 标记，在服务器端表示为一个 UIOutput 组件实例，相应的组件族名为 Output，相应的呈现器型名为 resource. Stylesheet。组件呈现为一个 HTML link 标记，用于在 JSF 页面中链接一个外部样式表。

在这里，样式表被看作是一种资源，相应的样式表文件必须被存放在 JSF 应用文档根目录下的 resources 目录下。通常可以在该目录下创建一些子目录，这些子目录称作库(library)，而资源则可存放在相应的库中。

假设 resources 目录下有一个名为 css 的子目录(库)，其中存放有一个名为 cssone. css 的样式表文件。那么应用中的任何一个 JSF 页面都可以用下面代码链接该外部样式表。

```
<h:outputStylesheet library="css" name="cssone.css"/>
```

其中，library 属性指定库名，name 属性指定资源名。

说明：JSF 提供对资源库或单个资源的版本控制机制。对于资源库的版本控制，可以在库目录下建立库的版本目录，而该库的不同版本的资源存放在相应的版本目录下。比如有以下目录和样式表文件：

```
resources/css/1_0/cssone.css
resources/css/1_1/cssone.css
```

那么，上述 h:outputStylesheet 标记会链接最新版本(1_1)库中的样式表文件 cssone. css。

对于单个资源的版本控制，应该将资源名作为库目录下的子目录名，而不同版本的资源文件则用版本号命名，比如：

```
resources/css/cssone.css/1_0.css
resources/css/cssone.css/1_1.css
```

那么，上述 h:outputStylesheet 标记会链接资源的最新版本，即样式表文件 1_1. css。版本号必须是下划线分隔的十进制数字，按正常的方式比较新旧。

6.1.3　CSS 应用示例

本应用项目(ch6_css)主要用于演示 CSS 样式表的定义与使用。应用包含两个层叠样式文件 css1.css 和 css2.css,两个 JSF 页面文件 index.xhtml 和 index1.xhtml。这 4 个文件的存储结构如图 6-1 所示。

这里,页面 index.xhtml 页面直接通过 HTML 的 link 标记链接 css1.css 样式表,而页面 index1.xhtml 则通过 JSF HTML 的 h:outputStylesheet 标记链接 css2.css 样式表。

ch6_css
Web 页
WEB-INF
css
css1.css
resources
css
css2.css
index.xhtml
index1.xhtml
源包

图 6-1　项目文件存储示意图

1. 页面 index.xhtml

该页面用到 css1.css 层叠样式表。先来看一下这个样式表的内容,见代码清单 6-1。样式表共定义了两个样式,两个样式都指定了(块级)元素的宽度、高度和内部文字的水平对齐方式。其中,第 1 个样式适用于标记 h3 和 class 属性值包含"b1"的标记;第 2 个样式适用于 class 属性值包含"b2"的标记 p。

代码清单 6-1　css/css1.css

```
h3,.b1{
  width: 400px;
  height: 30px;
  text-align: center;
}
p.b2{
  width: 400px;
  height: 30px;
  text-align: right;
}
```

页面 index.xhtml 本身也定义了一个样式表,见代码清单 6-2。其中,第 1 个样式指定了前景颜色和背景颜色,适用于 class 属性值包含"i1"标记;第 2 个样式指定了文字的字体,适用于 class 属性值包含"i2"的标记。

代码清单 6-2　index.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>CSS 应用示例(一)</title>
    <link rel="stylesheet" type="text/css" href="css/css1.css"/>
    <style type="text/css">
      .i1{
        color: white;
```

```
            background-color: gray;
          }
          .i2{
            font-family: 楷体_gb2312;
          }
        </style>
      </h:head>
      <h:body>
        <h3 class="i1">块级标记文字 1</h3>
        <span class="i1">行级标记文字 1</span>
        <h:outputText value="行级标记文字 2" styleClass="i2"/>
        <p class="b2 i1">块级标记文字 2</p>
        <div class="b1 i2">块级标记文字 3</div>
      </h:body>
    </html>
```

页面的呈现效果如图 6-2 所示。以最后一行内容为例,其效果是由外部样式表的第 1 个样式和内部样式表的第 2 个样式叠加的结果。



图 6-2　页面 index. xhtml 呈现效果

2. 页面 index1. xhtml

该页面用到 css2. css 层叠样式表。样式表中仅定义了一个样式,指定了前景颜色、背景颜色和文本对齐方式,见代码清单 6-3。

代码清单 6-3　resources/css/css2. css

```
.c{
  color: white;
  background-color: gray;
  text-align: right
}
```

页面 index1. xhtml(代码清单 6-4)主要演示块在其容器内的对齐方式,以及文字在块内的对齐方式。页面内共有两个大块,每个大块内都分别包含两个小块,每个小块内都有一段文本。

代码清单 6-4　index1.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>CSS 应用示例(二)</title>
    <h:outputStylesheet library="css" name="css2.css"/>
  </h:head>
  <h:body>
    <div style="margin: 10px 30px 0px 30px;border: solid gray">
      <div>
        <h:outputText value="1:默认"/>
      </div>
      <div class="c">
        <h:outputText value="1:右对齐"/>
      </div>
    </div>
    <div style="width:95%;margin: 10px auto 0px auto;border: solid gray">
      <div>
        <h:outputText value="2:默认"/>
      </div>
      <div class="c">
        <h:outputText value="2:右对齐"/>
      </div>
    </div>
  </h:body>
</html>
```

页面的呈现效果如图 6-3 所示。第 1 个大块边框与其上面标记或其容器上沿间距为 10px,与其容器右沿间距为 30px,与其下面标记间距 0px,与其容器左沿间距为 30px。这一大块内包含上下两个小块,两个小块的宽度是其容器的宽度。上面小块内的文字采用默认的左对齐,下面小块内的文字采用右对齐。

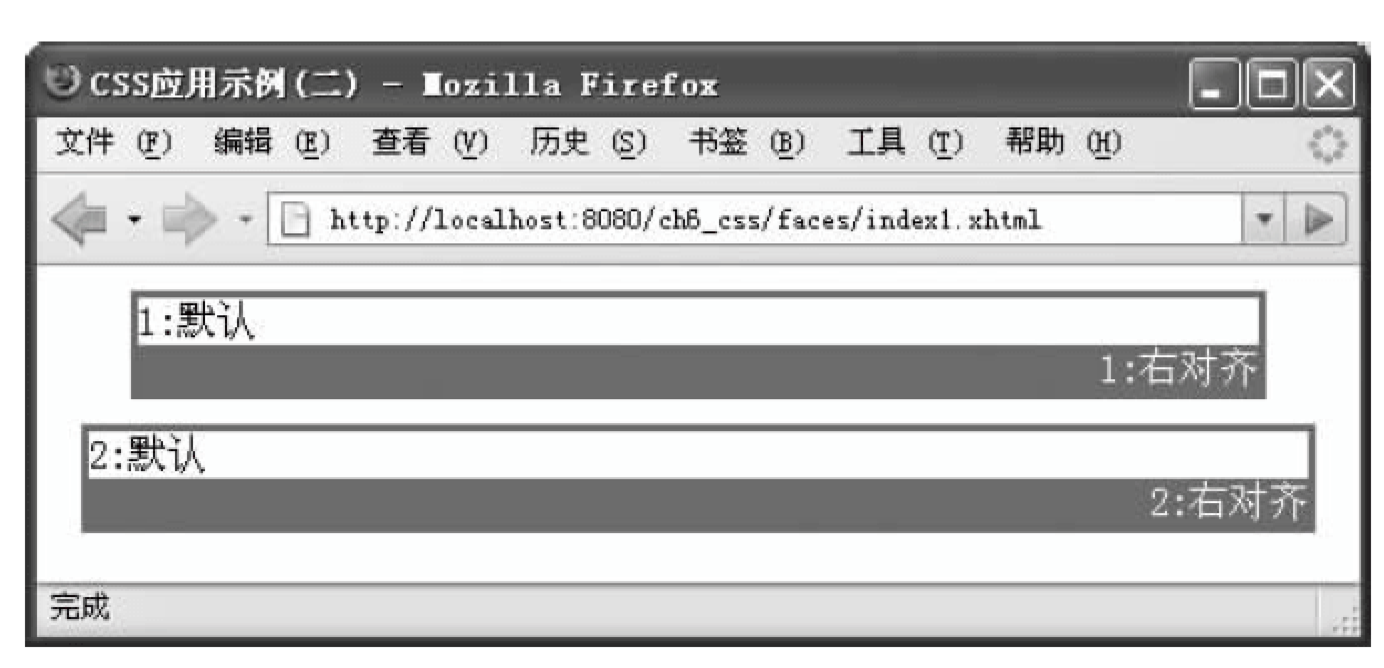


图 6-3　页面 index1.xhtml 呈现效果

第 2 个大块指定了其宽度是其容器宽度的 95%，其边框与其上面标记的间距为 10px、与其下面标记的间距为 0px。其边框与其容器左右边框的间距是 auto，这可以保证该块在其容器内可水平居中对齐。该大块内的情况与第 1 个大块内的情况类似。

说明：HTML 标记大致可分为块级标记和行内标记两大类。块级标记总是在新行上显示，可以设置高度(height)、宽度(width)，其默认宽度通常是它容器的宽度。块级标记可以包含其他的块级标记和行内标记。常见的块级标记如 div、form、p、table、h1 等。行内标记和其他行内标记显示在同一行上，它不能设置高度(height)和宽度(width)。行内标记通常只能包含文本和其他行内标记。常见的行内标记如 a、br、input 等。

6.2 面　　板

面板类标记主要用于布局。面板类标记的相应组件类有共同的超类 UIPanel，在该超类中定义了这些组件共同的组件族名 Panel。

6.2.1 h:panelGrid 标记

该标记在服务器端表示为一个 HtmlPanelGrid 组件实例，其呈现器型名为 Grid。组件呈现为一个 HTML table 元素，形成一个由单元格组成的表格，每个单元格可以放置一个子组件。

在标记中，可以用 columns 属性指定表格的列数，如：

```
<h:panelGrid columns="3">
    ……
</h:panelGrid>
```

表格的行数由 columns 属性指定的列数和子组件的数目共同决定。比如对于上述 3 列的表格，假定共有 10 个子组件，那么表格呈现为 4 行，其中最后一行只有一个子组件。各子组件将被依次从上向下、从左到右放置在表格内。

h:panelGrid 标记的属性较多，除 id、rendered、style、styleClass 和 binding 等基本属性外，还包含许多反映表格特点的属性。下面介绍 h:panelGrid 标记的常用属性。

(1) columns：设置表格的列数，int 型。默认值为 1。

(2) width：设置整个表格的宽度，String 型。一般块级元素的默认宽度是其容器的宽度，但表格的默认宽度则由其各数据元素的宽度共同决定。

(3) bgcolor：设置表格的背景颜色，String 型。

(4) cellpadding：指定单元格内空白，即单元格边界与单元格内容之间的间距，String 型。

(5) cellspacing：指定单元格间空白，既指单元格与单元格之间的间距，也指表格边界与单元格之间的间距，String 型。

(6) frame：指定表格四条边框线的可视性，String 型。该属性的有效值包括：

- none：无边框线(默认值)。
- above：仅有顶框。

- below：仅有底框。
- hsides：仅有顶框和底框。
- lhs：仅有左侧框。
- rhs：仅有右侧框。
- vsides：仅有左侧框和右侧框。
- box、border：包含全部四条边框。

（7）rules：指定表格内单元格边框的可视性，String 型。该属性的有效值包括：

- none：无分隔线（默认值）。
- groups：仅在行组和列组间画分隔线。
- rows：仅有行分隔线。
- cols：仅有列分隔线。
- all：包括所有分隔线。

说明：相邻的边框线的间距由 cellspacing 属性指定。相邻的边框线也可以合二为一，这时需要用到 CSS 属性 border-collapse。若该属性设置为 separate，则相邻边框线是分离的；若属性值取 collapse，那么相邻边框线将合二为一。

（8）border：设置表格的边框宽度（以像素为单位），int 型。默认值为 1。

该属性的设置也会影响 frame 和 rules 属性的取值。当 border 属性设置为 0 时，将意味着 frame＝"none"，除非特别设置，否则 rules＝"none"。当 border 属性设置为非 0 时，除非特别设置，否则意味着 frame＝"border"和 rules＝"all"。

（9）columnClasses：指定作用于列的用逗号分隔的 CSS 样式类的列表，列表中的每个样式类依次作用于表格中的列。

可以对一列应用多个样式类，这些样式类之间用空格分隔。

（10）rowClasses：指定作用于行的用逗号分隔的 CSS 样式类的列表，列表中的每个样式类依次作用于表格中的行。如果样式类的数目少于表格行数，那么这些样式类将被循环重复使用。

可以对一行应用多个样式类，这些样式类之间用空格分隔。

6.2.2 h:panelGroup 标记

用于将一些组件归组，以便它们被看作是一个组件。

h:panelGroup 标记经常与 h:panelGrid 标记一起使用。在表格中，一个单元格通常只能包含一个组件。如果要包含多个组件，可以用 h:panelGroup 标记对它们进行归组。反过来，如果需要一个空白单元格，也可以用一个不含任何子标记的 h:panelGroup 标记填充之。

该标记的属性较少。除 id、style、styleClass、rendered 和 binding 等其他大多数标记都具有的基本属性外，该标记的一个主要属性是 layout。

该标记在服务器端表示为一个 HtmlPanelGroup 组件实例，其呈现器型名为 Group。

如果没有指定 style 或 styleClass 属性，组件呈现时将直接呈现内部子组件。如果指定了 style 或 styleClass 属性、且 layout 属性取 block，组件呈现一个 HTML div 标记；如果指定了 style 或 styleClass 属性、而 layout 属性取非 block 值，组件呈现一个 HTML span 标记。

6.3 数据表格

本节介绍如何在页面中使用数据表格，包括用数据表格显示一个数据集，设置表格的标题、头和列，以及编辑表格数据等技术。

6.3.1 用数据表格显示数据集

用数据表格显示数据集，主要涉及 h:dataTable 标记和 h:column 标记。

h:dataTable 标记在服务器端表示为一个 HtmlDataTable 组件实例，其组件族名为 UIData，呈现器型名为 Table。组件依据要显示的数据集以及相应的子组件呈现一个数据表格，即一个 HTML table 标记。

h:column 标记在服务器端表示为一个 HtmlColumn 组件实例，其组件族名为 UIColumn。这种组件没有相应的呈现器，而是作为 HtmlDataTable 组件的子组件，由其父组件相关联的呈现器统一呈现。

下面代码演示了 h:dataTable 标记和 h:column 标记的用法。

```
<h:dataTable value="#{bean.books}" var="book">
    <h:column>
        <h:outputText value="#{book.title}"/>
    </h:column>
    <h:column>
        <h:outputText value="#{book.price}"/>
    </h:column>
</h:dataTable>
```

h:dataTable 标记最重要的属性是的 value 和 var 属性。value 属性指定要显示的数据集。这个数据集可以是下列某种形式：

- 一个 Java 对象。
- 一个数组。
- 一个 java.util.List 对象。
- 一个 javax.faces.model.DataModel 对象。
- 一个 java.sql.ResultSet 对象。
- 一个 javax.servlet.jsp.jstl.sql.ResultSet 对象。
- 一个 javax.sql.RowSet 对象。

虽然 value 属性可以引用单个对象，但通常引用的是一些对象的集合或者是数据库查询的结果集，如 List 对象、ResultSet 对象等。当标记组件呈现时，将迭代处理集合中的每个元素。每个元素通常被呈现为数据表格中的一行。

var 属性指定一个请求作用域的名称，用于表示当前被处理的元素对象，就如一个 bean 名称表示一个 bean 实例。该名称可用于 h:dataTable 标记的子标记内的 EL 表达式中。

除了 value 和 var 属性，h:dataTable 标记的常用属性还包括：

(1) first：指定从数据集的第几个元素(索引值从 0 开始)开始迭代处理和呈现。默认

值为 0,表示从首行开始处理和呈现。

(2) rows：指定要处理和呈现的元素个数(行数)。若为 0(默认值),则处理和呈现数据集中的所有元素。

(3) captionStyle：指定表标题的 CSS 样式。

(4) captionClass：指定表标题的 CSS 样式类。

(5) headerClass：指定表头的 CSS 样式类,多个样式类用空格分隔。该样式类作用于表中所有表头。

(6) footerClass：指定表脚的 CSS 样式类,多个样式类用空格分隔。该样式类作用于表中所有表脚。

另外,h：panelGrid 标记中介绍的除 columns 属性外的其他属性也都可用在 h:dataTable 标记中。

h:dataTable 标记中一般会嵌套若干个 h:column 子标记,用于配置数据表格中的列。每个 h:column 标记扮演了特定列的模板,其内容对将要处理和呈现的每个元素对象进行重复。在 h:column 标记内,一般需要使用 h:dataTable 标记的 var 属性指定的名称,通常在 EL 表达式中通过该名称访问当前元素对象的某个数据。

h:column 标记的属性较少,除 id、rendered 和 binding 等基本属性外,主要有以下两个属性：

(1) headerClass：指定该列表头的 CSS 样式类。

(2) footerClass：指定该列表脚的 CSS 样式类。

6.3.2 标题、表头和表脚

上面 h:dataTable 和 h:column 两个标记的有关属性都涉及表标题或表头和表脚的样式设置,这里介绍如何设置表标题、表头和表脚本身。

表标题通常显示于表格上方,不在表格的单元格内显示。表头显示于表格头部,在表格单元格内显示。表脚显示于表格尾部,在表格单元格内显示。

标题、表头和表脚都是通过 f:facet 核心标记来配置的。f:facet 标记用来为其父组件和其子组件之间申请一种特殊的关系,其仅有的 name 属性用于指定这种关系。

下面代码演示了设置表标题的方法。其中 f:facet 标记应该直接放置在 h:dataTable 标记内,其 name 属性取“caption”,表明其子标记(这里是 h:outputText 标记)将呈现为其所在表格的标题。

```
<h:dataTable …>
    <f:facet name="caption">
        <h:outputText value="表标题"/>
    </f:facet>
    ……
</h:dataTable>
```

下面代码演示了设置表头和表脚的方法。要设置表头,f:facet 标记的 name 属性应设置为“header”;要设置表脚,f:facet 标记的 name 属性应设置为“footer”。

```
<h:dataTable value="#{bean.books}" var="book">
```

```
    <h:column>
        <f:facet name="header">
            <h:outputText value="列头"/>
        </f:facet>
        #{book.title}
        <f:facet name="footer">
            <h:outputText value="列脚"/>
        </f:facet>
    </h:column>
    ……
</h:dataTable>
```

上面设置表头和表脚的 f:facet 标记都放置在 h:column 标记内，这样的表头、表脚也称为列头、列脚，分别显示于所在列的第一个单元格和最后一个单元格。所有的列头统称为表头，所有的列脚统称为表脚。

用于设置表头和表脚的 f:facet 标记也可以放置在 h:dataTable 标记内，h:column 标记外，这样的表头、表脚将横跨表格所有的列。

在呈现一个表格时，首先呈现表的标题，然后依次呈现横跨所有列的表头、从属于某列的表头，接着迭代处理和呈现数据集中元素对象，最后依次呈现从属于某列的表脚、横跨所有列的表脚。

6.3.3 编辑表格

上面介绍了用数据表格显示数据集数据的方法，一般只需在 h:column 标记内放置 EL 表达式或 h:outputText 子标记，利用它们来显示数据集中当前元素对象的某个数据。

实际上，不仅可以用数据表格显示数据集数据，也可以用数据表格编辑数据集数据。要用数据表格编辑数据集数据，需要在 h:column 标记内放置 h:inputText 等输入类组件标记，以便能显示和编辑数据集数据，然后再通过动作按钮或超链接提交这些数据。当然，这些输入类组件和动作按钮或超链接都应该放置在表单内。

另外，也可以在表格中设置一个按钮或超链接列，即表格中的每一行都有一个按钮或超链接。这样，单击某个按钮或超链接就可以对所在的行(元素对象)进行特定的操作。

下面通过一个应用示例介绍表格数据的编辑。该 JSF 应用项目(ch6_editanddelete)包含一个 JSF 页面、一个样式表文件、一个受管 bean 和两个普通的 Java 类。项目运行时页面的呈现效果如图 6-4 所示。

页面显示购物车的内容。购物车中每个选购项的数量可以修改，当单击“保存”按钮时，修改的结果将被保存到购物车中。当单击“删除”超链接时，所在的选购项将从购物车中删除，同时其他选购项的修改后的数量也会进行保存。

1. JSF 页面

该应用项目唯一的 JSF 页面 index.xhtml 如代码清单 6-5 所示。页面主要包含一个放置在表单内的数据表格。数据表格标记(h:dataTable)的 value 属性绑定至受管 bean(名为 tableData)的 cart 属性。cart 属性的类型为 List<Item>，代表一个购物车，其中每个元素的类型为 Item，代表一个选购项。



图 6-4 应用 ch6_editanddelete 运行效果图

代码清单 6-5 index.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
 <h:head>
   <title>ch6_editanddelete</title>
   <h:outputStylesheet library="css" name="style.css"/>
 </h:head>
 <h:body>
   <h:form>
     <h:dataTable value="#{tableData.cart}" var="item" headerClass="header"
                    styleClass="data">
     <h:column>
       <f:facet name="header">书名</f:facet>
       <h:outputText value="#{item.book.title}"/>
     </h:column>
     <h:column>
       <f:facet name="header">单价</f:facet>
       <h:outputText value="#{item.book.price}"/>
     </h:column>
     <h:column>
       <f:facet name="header">数量</f:facet>
       <h:inputText value="#{item.num}" styleClass="field"/>
     </h:column>
     <h:column>
       <h:commandLink value="删除" action="#{tableData.delete(item)}"/>
     </h:column>
     <f:facet name="footer">
         <h:commandButton value="保存"action= "index?faces-redirect=true"/>
     </f:facet>
```

```
        </h:dataTable>
        </h:form>
    </h:body>
</html>
```

这里，“删除”超链接指定了一个带参数的方法表达式，而“保存”按钮并没有指定方法表达式。当单击“保存”按钮时，所有的表单数据(即各选购项的数量)被收集并提交。JSF 框架在处理请求时，这些数据将被读入、转换、验证，并更新模型值(即各选购项的数量)，最后再次呈现该页面。

2. 样式表文件

该应用项目仅由一个样式表文件，如代码清单 6-6 所示。其中样式类 data 用于表格内普通数据内容的显示，样式类 header 用于表格内列头的显示，样式类 field 用于表格内文本域的显示。

代码清单 6-6　resources/css/style.css

```
.data {
  font-size: 14px
}
.header {
  text-align: left;
  font-size: 14px
}
.field {
  width: 20px;
  font-size: 13px;
}
```

3. 受管 bean

本应用仅包含一个受管 bean 类，文件名为 TableData.java(代码清单 6-7)。这是一个会话作用域的受管 bean，包含一个只读属性 cart(购物车)，属性值在 bean 实例被创建时初始化。该受管 bean 还包含一个带参数的动作方法 delete(Item)，会在用户单击页面上的“删除”按钮时被调用。

代码清单 6-7　TableData.java

```
package bean;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import source.Book;
import source.Item;
import java.io.Serializable;
@ManagedBean
@SessionScoped
```

```
public class TableData implements Serializable {
  private List<Item>cart=new ArrayList<Item>();

  public TableData(){
    Item item;
    item=new Item(new Book("01","计算机原理",new BigDecimal("29.50")),1);
    cart.add(item);
    item=new Item(new Book("02","数据库原理",new BigDecimal("35.00")),1);
    cart.add(item);
    item=new Item(new Book("03","计算机操作系统",new BigDecimal("31.50")),2);
    cart.add(item);
    item=new Item(new Book("04","Java 程序设计",new BigDecimal("33.50")),1);
    cart.add(item);
  }

  public List<Item>getCart(){
    return cart;
  }
  public String delete(Item item){
    cart.remove(item);
    return "index?faces-redirect=true";
  }
}
```

4. Java 类

该应用项目包含两个普通的 Java 类,都存放在 source 包中。类 Book(代码清单 6-8)表示图书,包含 3 个只读属性,即 bookid(编号)、title(书名)和 price(单价)。类 Item(代码清单 6-9)表示选购项,包含一个可读写属性 num(数量)和一个只读属性 book(图书)。

代码清单 6-8　Book.java

```
package source;
import java.io.Serializable;
import java.math.BigDecimal;

public class Book implements Serializable {
  private String bookid;
  private String title;
  private BigDecimal price;

  public Book(){
  }
  public Book(String bookid,String title,BigDecimal price){
    this.bookid=bookid;
    this.title=title;
    this.price=price;
  }
```

```
    public String getBookid(){
      return bookid;
    }
    public String getTitle(){
      return title;
    }
    public BigDecimal getPrice(){
      return price;
    }
}
```

代码清单 6-9　Item.java

```
package source;
import java.io.Serializable;

public class Item implements Serializable{
    private Book book;
    private int num;

    public Item(Book book,int num){
      this.book=book;
      this.num=num;
    }

    public int getNum(){
      return num;
    }
    public void setNum(int num){
      this.num=num;
    }
    public Book getBook(){
      return book;
    }
}
```

在该例中，表格中“数量”列的各单元格是 h:inputText 标记，其 value 属性指定一个值表达式；表格最后一列的各单元格是 h:commandLink 标记，其 action 属性指定一个方法表达式。下面讨论一下这些值表达式和方法表达式的计算。

在呈现数据表格时，对于数据集(购物车)中的每个元素对象(选购项)都显示对应的表格行，其中“数量”列的单元格会包含一个 HTML input 标记，EL 表达式“#{item.num}”以右值模式被计算，结果当前选购项的数量将作为 input 标记的值显示于单元格内。这里，input 标记会与当前选购项的索引值相关联。删除列的单元格显示一个功能与提交按钮一样 HTML a 标记，并且与当前选购项的索引值相关联。

当单击“保存”按钮时，所有的“数量”值被提交，其中每个“数量”值与一个索引值相关联。在服务器端，JSF 框架会根据索引值找到购物车中相应的选购项，然后以左值模式计算 EL 表达式“#{item.num}”，即用接收到的“数量”值去更新选购项的对应属性。

当单击某个“删除”超链接时，同样会对“数量”值进行提交，并更新模型值。另外，它会在“调用应用”阶段调用 tableData 受管 bean 的 delete 方法。此时，JSF 框架会根据与此“删除”超链接相关联的索引值定位选购项，并用该选购项作为方法调用的参数。

6.4 论坛—主题表与回复表

本节继续 5.8 节介绍的应用项目(luntan_input)，对其进行功能扩充并做必要的修改。为保持之前项目的独立性，这里新创建一个名为 luntan_table 的 JSF 应用项目，并从原先项目复制所有的 JSF 页面、源包中的所有 Java 包和 Java 类。

与原先的应用项目 luntan_input 相比，新的应用项目 luntan_table 主要增加了以下功能：在主页上增加主题表、并对整个页面进行布局和格式化；在“查看回复”页面上增加回复表、并对整个页面进行布局和格式化。

图 6-5 显示了该应用的运行结果。图 6-5(a)是主页页面，页面原有的元素都还存在，只是进行了重新布局和格式化。页面下方添加了一个主题表，每个主题的标题是一个超链接。



(a)

(b)

图 6-5 应用 luntan_table 运行效果图

单击某主题的标题超链接，会在一个新窗口打开“查看回复”页面。图 6-5(b)是“查看回复”页面，该页面同样保留了原有的元素，但进行了重新布局和格式化。页面下方是新添加的回复表，回复表的最前面是指定主题的内容本身，其后才是对该主题的回复。

6.4.1 扩充模型和受管 bean

从图 6-5 可以看出，主题表和回复表都要显示主题或回复的创建时间。这个时间数据的类型是 java.util.Date，在显示之前一般要进行相应的格式化。另外，回复内容是在一个文本区内输入的，期间换行是通过按 Enter 键实现的。按 Enter 键相当于插入了回车换行符。而在回复表中，回复内容是在表格的一个单元格内显示的，其中回车换行符并不能呈现为换行的效果，因此在显示之前需要有一个转换的过程。

为此在应用的 util 包中新建一个 Formater 类，其中的两个静态方法分别用于支持完成上述格式化和转换任务，见代码清单 6-10。

代码清单 6-10　Formater.java

```
package util;
import java.text.DateFormat;
import java.util.Date;

public class Formater {
  public static String getTime(Date date){
    return DateFormat.getDateTimeInstance(DateFormat.MEDIUM,
                             DateFormat.SHORT).format(date);
  }
  public static String getContent(String content){
    String str=content.replaceAll("\r\n","<br/>");
    return str;
  }
}
```

从图 6-5 还可以看出，每个主题都有一个业务数据：点击数。当用户单击主题表中某主题的标题超链接时，主题的点击数就应该增 1。为此，在应用的 model 包中的 TopicManager 类中添加一个业务方法 clickTopic(Topic)，见代码清单 6-11。当用户单击主题时，应调用该方法完成相应的业务处理。

代码清单 6-11　TopicManager 类中的 clickTopic(Topic)方法

```
public void clickTopic(Topic topic){                    //主题的点击数增 1
  topic.setClickcount(topic.getClickcount()+1);
}
```

最后需要扩充受管 bean，它扮演着连接页面与模型的桥梁作用。作为支撑主页的受管 bean，bean.Index 类需要添加 showTime(Date)方法和 click()方法，见代码清单 6-12。主页可以通过一个 EL 方法表达式调用 showTime 方法实现时间数据的格式化。click 是一个动作方法，在用户单击某主题的标题时调用。

代码清单 6-12　为 bean. Index 类新添加的方法

```
public String showTime(Date date){                          //格式化时间
  return Formater.getTime(date);
}
public String click(){
  int tid=Integer.parseInt((String)ELUtil .evalEL("#{param.tid}"));
  TopicManager tm=new TopicManager();
  Topic t=tm.getTopicById(tid);
  tm.clickTopic(t);
  return "showReply?faces-redirect=true&amp;tid="+tid;
}
```

作为支撑“查看回复”页面的受管 bean，bean. ShowReply 类需要添加 showTime(Date) 和 showContent(String)方法，见代码清单 6-13。“查看回复”页面可以通过 EL 方法表达式调用这两个方法完成时间数据的格式化和回复内容的转换。

代码清单 6-13　bean. ShowReply 类新添加的方法

```
public String showTime(Date date){        //格式化时间数据
  return Formater.getTime(date);
}
public String showContent(String content){ //转换内容：将其中的回车换行替换成<br/>
  return Formater.getContent(content);
}
```

6.4.2　创建样式表

该应用新建了一个外部样式表文件 style. css，见代码清单 6-14。样式表主要根据主页和“查看回复”两个页面的布局和格式化需要建立，定义了它们所需的一些公共样式。有些标记所需的特殊显示格式则采用内联样式，在标记的 style 属性中直接指定。

代码清单 6-14　resources/css/luntan_css. css

```
.font1 {font-size:14px}
.font2 {font-size:14px;font-family:黑体;font-weight:normal}
.font3 {font-size: 12px}
.text_center {text-align:center}
.text_right {text-align:right}
.text_left {text-align:left}
.bg_color {background-color:lightsteelblue}
.margin10 {margin:10px auto 0px auto}
.margin0 {margin:0px auto 0px auto}
.width100 {width:100%}
.width95 {width:95%}
.word_break {word-wrap:break-word;word-break:break-all;overflow:hidden}
a:link {color:teal;text-decoration:none}
a:visited {color:teal;text-decoration:none}
a:hover {color:red;text-decoration:underline}
```

6.4.3 修改主页

对主页 index.xhtml 的修改，首先要用 h:outputStylesheet 标记链接外部样式表，然后对页面原有的元素进行格式化，最后在页面的尾部添加主题表。

该页面布局的基本思路是：整个页面由一些从上到下排列（间距为 0 或 10px）、宽度一样（窗口宽度的 95%）、水平居中对齐的块组成。

与上一个项目的主页相比，该页面虽然保留了原有的元素，但由于格式化的需要，页面中几乎所有的标记都有所改动。为体现整体性，代码清单 6-15 给出了该页面的所有代码。

代码清单 6-15　主页（index.xhtml）

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
 <h:head>
  <title>轻松论坛--主页</title>
  <h:outputStylesheet library="css" name="luntan_css.css"/>
 </h:head>
 <h:body>
  <h:form styleClass="margin10 width95">
   <h:panelGrid columns="2" cellspacing="5px" styleClass="font1 width100">
    <h:panelGroup layout="block" styleClass="text_left">
     <h:outputText value="注册人数:#{index.numOfClient},"/>
     <h:outputText value="在线人数:#{index.numOfOnline}"/>
    </h:panelGroup>
    <h:panelGroup layout="block" styleClass="text_right">
     <h:commandLink value="登录" action="login"
                 rendered="#{sessinfo.client==null}"/>
     <h:outputText value=" "/>
     <h:commandLink value="注册" action="registry"
                 rendered="#{sessinfo.client==null}"/>
     <h:outputText value="#{sessinfo.client.username}:"
                 rendered="#{sessinfo.client!=null}"/>
     <h:commandLink value="退出" action="#{index.exit}"
                 rendered="#{sessinfo.client!=null}"/>
    </h:panelGroup>
   </h:panelGrid>
  </h:form>

  <h:form styleClass="margin0 width95">
```

```
      <h:panelGrid columns="2" cellspacing="5px" styleClass="font1 width100">
        <h:panelGroup layout="block" styleClass="text_left">
          <h:outputText value="主题数:#{index.topics.size()}"/>
        </h:panelGroup>
        <h:panelGroup layout="block" styleClass="text_right">
          <h:commandButton value="新建主题" action="inputTopic?faces-redirect=
              true" styleClass="font3" disabled="#{sessinfo.client==null}"/>
        </h:panelGroup>
      </h:panelGrid>
    </h:form>

    <h:form styleClass="margin0 width95">
      <h:dataTable value="#{index.topics}" var="topic" cellspacing="5px"
          styleClass="font1 width100"
          headerClass="text_left font2 bg_color">
        <h:column>
          <f:facet name="header"><h:outputText value="标题"/></f:facet>
          <h:commandLink value=
              "#{topic.title.length()>20?topic.title.substring(0,20):topic.title}"
              action="#{index.click}" title="#{topic.title}" target="reply">
            <f:param name="tid" value="#{topic.id}"/>
          </h:commandLink>
        </h:column>
        <h:column>
          <f:facet name="header"><h:outputText value="回复/点击"/></f:facet>
          <h:outputText value="#{topic.replycount}/#{topic.clickcount}"/>
        </h:column>
        <h:column>
          <f:facet name="header"><h:outputText value="创建者/时间"/></f:facet>
          <h:outputText value="#{topic.client.username}/"/>
          <h:outputText value="#{index.showTime(topic.createtime)}"
                    styleClass="font3"/>
        </h:column>
        <h:column>
          <f:facet name="header"><h:outputText value="最后回复者/时间"/></f:facet>
          <h:outputText value="#{topic.client1.username}/"/>
          <h:outputText value="#{index.showTime(topic.lastreplytime)}"
                    styleClass="font3"/>
        </h:column>
      </h:dataTable>
    </h:form>
  </h:body>
</html>
```

6.4.4 修改“查看回复”页面

对“查看回复”页面 showReply.xhtml 的修改，其过程和布局思路与修改主页的基本相同。代码清单 6-16 给出了该页面的所有代码。

代码清单 6-16 showReply.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <f:metadata>
    <f:viewParam name="tid" value="#{showReply.tid}"/>
    <f:event type="preRenderView" listener="#{showReply.initView}"/>
  </f:metadata>
  <h:head>
    <title>轻松论坛--查看回复</title>
    <h:outputStylesheet library="css" name="luntan_css.css"/>
  </h:head>
  <h:body>
    <h:form styleClass="margin10 width95">
      <h:panelGrid columns="2" cellspacing="5px" styleClass="font1 width100">
        <h:panelGroup layout="block" styleClass="text_left">
          <h:outputText value="回复数:#{showReply.replys.size()-1}"/>
        </h:panelGroup>
        <h:panelGroup layout="block" styleClass="text_right">
          <h:commandButton value="回复主题" action="#{showReply.goReply}"
                styleClass="font3" disabled="#{sessinfo.client==null}"/>
        </h:panelGroup>
      </h:panelGrid>
    </h:form>

    <h:panelGroup layout="block" styleClass="margin0 width95">
      <h:panelGroup layout="block" styleClass="bg_color font1"
                          style="margin-left:5px;margin-right:5px">
        <h:outputText value="主题:#{showReply.topic.title}"/>
      </h:panelGroup>
    </h:panelGroup>

    <h:panelGroup layout="block" styleClass="margin0 width95">
      <h:dataTable value="#{showReply.replys}" var="reply" cellspacing="5px"
          styleClass="font1 width100">
      <h:column>
        <h:panelGrid columns="2" cellspacing="0px" cellpadding="0px"
```

```
                    styleClass="font3 width100">
                <h:panelGroup layout="block">
                  <h:outputText value="作者:#{reply.client.username}"/>
                  <h:outputText value="发表于:#{showReply.showTime(reply.replytime)}"/>
                </h:panelGroup>
                <h:panelGroup layout="block" styleClass="text_right">
                  <h:outputText value="##{showReply.replys.indexOf(reply)+1}"/>
                </h:panelGroup>
              </h:panelGrid>
              <h:panelGroup layout="block" styleClass="word_break" style="margin-top: 8px">
                <h:outputText value="#{showReply.showContent(reply.content)}"escape="false"/>
              </h:panelGroup>
              <hr/>
            </h:column>
            </h:dataTable>
        </h:panelGroup>
    </h:body>
</html>
```

6.5 论坛—分页显示

在 Web 应用中，当要显示大型数据集时，通常会利用数据表格进行分页显示。也就是说，每次传送数据集的一部分数据(即一页数据)至客户端，由数据表格组件进行呈现，并提供一系列页码超链接或按钮。当用户单击这些超链接或按钮时，将转而呈现上一页、下一页或指定页的数据。这种浏览方式直观、易用，已广泛用于各种 Web 应用。

在 JSF 的 h:dataTable 标记中，包括 first 和 rows 两个属性，它使得标记能够仅显示数据集的一部分数据，而不必呈现数据集的所有元素。这可以作为实现分页显示功能的基础。为实现分页显示功能，通常需要引入以下变量：

- pageSize：页面大小，即一次显示几个元素(行)。
- pageCount：总页数，可以根据数据集的大小与 pageSize 计算获得。
- currentPage：当前页码。该变量的初值可以设置为 1，之后能由用户指定。

其中，数据集的大小一般可以调用其 size()方法获得。这样，当用户指定某个当前页码时，很容易就可以计算出所需要的 first 和 rows 属性值：

```
first=pageSize*(currentPage-1)
rows=pageSize
```

下面继续 6.4 节介绍的应用项目(luntan_table)，对其进行功能扩充并做必要的修改。为保持之前项目的独立性，这里新创建一个名为 luntan_pagination 的 JSF 应用项目，并从原先项目复制所有的样式表文件、JSF 页面、源包中的所有 Java 包和 Java 类。

与原先的应用项目 luntan_table 相比，新的应用项目 luntan_pagination 主要为主页中的主题表和“查看回复”页面中的回复表增加了分页显示的功能。图 6-6 显示了该应用的运行结果，其中：图 6-6(a)是主页的呈现效果，图 6-6(b)是“查看回复”页面的呈现效果。

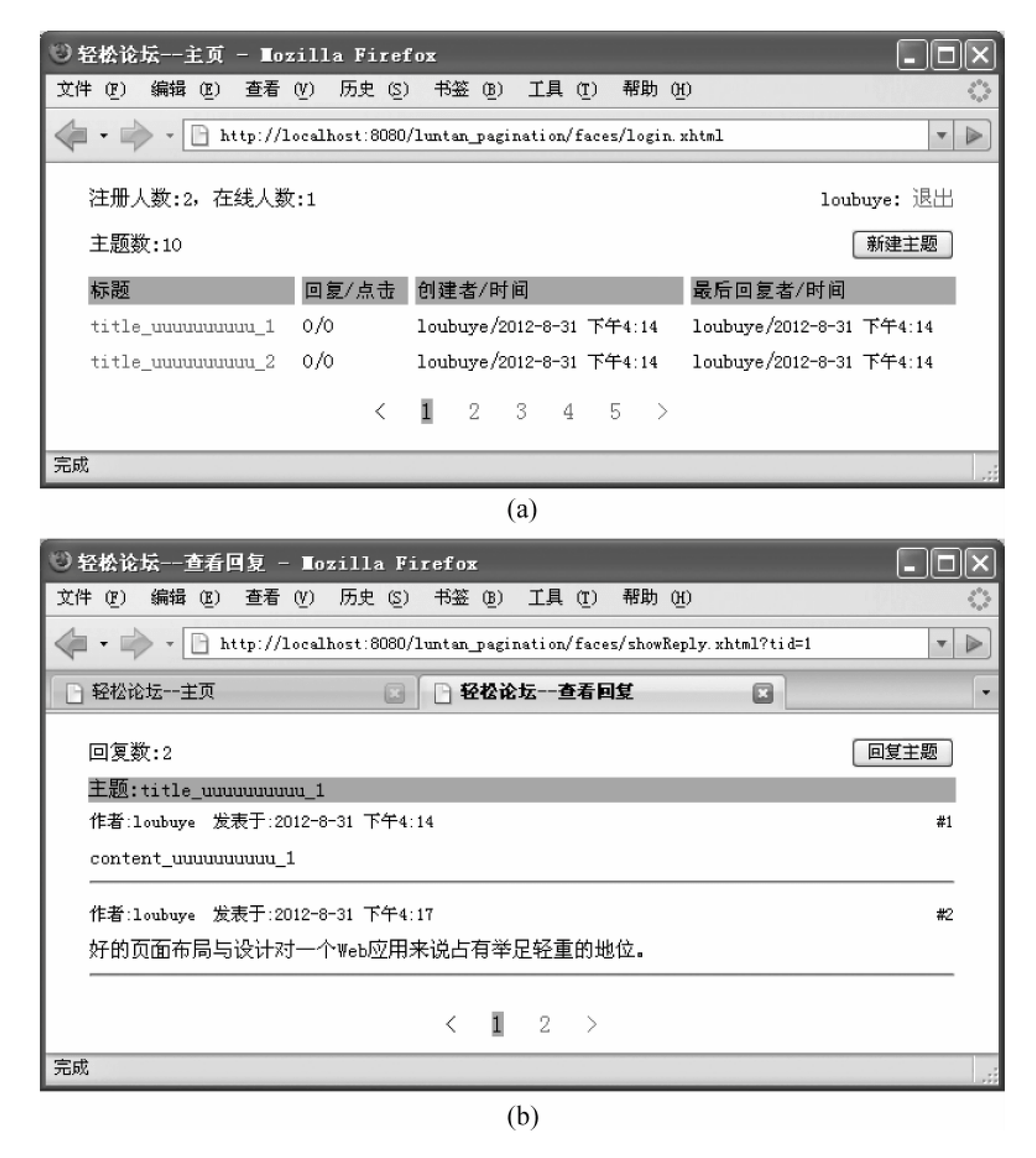


图 6-6 应用 luntan_pagination 运行效果图

6.5.1 创建辅助类

首先创建一个 Pager 辅助类，集中定义涉及分页显示功能的数据处理。该类定义在应用的 util 包中，具体代码如代码清单 6-17 所示。

代码清单 6-17 util.Pager

```
package util;
import java.util.ArrayList;
import java.util.List;

public class Pager {
  private int pageCount;                          //总页数
  private int currentPage=1;                      //当前页码
  private int showPages;                          //连续页码超链接数
  private List<Integer>pages;

  public void setPageCount(int pageCount){
    this.pageCount=pageCount;
```

```java
    }
    public int getPageCount(){
      return pageCount;
    }
    public void setShowPages(int showPages){
      this.showPages=showPages;
    }
    public int getCurrentPage(){
      return currentPage;
    }
    public void setCurrentPage(int currentPage){
      this.currentPage=currentPage;
    }
    public List<Integer>getPages(){
      return pages;
    }
    //根据当前页码、总页数和连续页码超链接数构建用于产生页码超链接的页码表
    public void init(){
      if(pageCount>0){
        if(currentPage<1) currentPage=1;
        if(currentPage>pageCount) currentPage=pageCount;
        int startPage=currentPage-(showPages-1)/2;       //初步的最小页码
        int endPage=currentPage+showPages/2;             //初步的最大页码
        int s=0;                                         //最小页码的偏差
        if(startPage<1){
          s=-startPage+1;
          startPage=1;
        }
        int e=0;                                         //最大页码的偏差
        if(endPage>pageCount){
          e=endPage-pageCount;
          endPage=pageCount;
        }
        startPage=Math.max(startPage-e,1);       //用最大页码的偏差去调用最小页码
        endPage=Math.min(endPage+s,pageCount);   //用最小页码的偏差去调用最大页码
        pages=new ArrayList<Integer>();
        for(int i=startPage;i<=endPage;i++){
          pages.add(i);
        }
        System.out.println(startPage+","+endPage);
      }
    }
}
```

在该应用中，分页显示功能主要是为用户提供一系列的超链接，用户单击超链接可以指定所需数据页的页码。这些超链接除包括前(<)、后(>)、第 1 页页码和最后一页页码，还

包括含当前页码在内的连续的若干个页码,这个连续的若干个页码超链接的数量是可以设置的。这里,当前页码不呈现为超链接;如果当前页是第 1 页,那么“前(<)”不呈现为超链接;如果当前页是最后一页,那么“后(>)”不呈现为超链接。

该类的 init 方法会根据当前页码、总页数和连续页码超链接数构建相应的页码表。该页码表应包含当前页码。在页面呈现时,当前页码会呈现为普通文本,而其他页码则呈现为超链接。

Pager 类不包含数据集,也不包括页面大小(pageSize)。这些属性放置在支撑页面呈现的受管 bean 中。一个需要分页显示数据集的页面的支撑 bean 应该扩展 Pager 类。

6.5.2 修改主页

首先需要修改支撑主页呈现的受管 bean 类,即 Index.java。对它的修改包括以下几个方面:

(1) 将 bean 的作用域改为会话作用域。

(2) 让该 bean 类扩展 util.Pager 类。

(3) 增加一个只读的 pageSize 属性。该属性的值可以根据需要设置。根据该属性和数据集的大小可以计算出总页数。

(4) 增加一个私有的 loadData()方法,用于装入主题数据。同时修改构造方法。

(5) 增加 initView()方法。该方法是一个 preRenderView 事件监听方法,其功能是:装入主题数据,设置 pageCount 和 showPages 属性,然后调用 init 方法构建页码表。

代码清单 6-18 列出该类所作的变动。

代码清单 6-18　bean.Index(部分)

```
@ManagedBean
@SessionScoped
public class Index extends Pager implements Serializable{
  public Index(){
    loadData();
  }
  private void loadData(){
    TopicManager tm=new TopicManager();
    topics=tm.getTopics();
  }
  ......
  private int pageSize=2;                 //可根据需要设置初值
  public int getPageSize(){
    return pageSize;
  }
  public void initView(){
    if(getCurrentPage()==1) loadData();
    int pc=(int)Math.ceil(topics.size()*1.0/pageSize);
    setPageCount(pc);
    setShowPages(8);                      //设置 showPages 属性
```

```
        init();
    }
}
```

如果分页显示数据集且要对显示的数据元素做某些操作,如通过文本域进行编辑修改、通过动作类组件产生动作事件,那么一般应该将其对应的受管 bean 设置为会话作用域,以便该受管 bean 在请求时的状态与之前页面呈现时的状态相同。这样,请求参数才能被正确接收和处理。

由于该受管 bean(bean.Index)被设置成会话作用域了,所以在整个会话期间,该 bean 实例只被创建一次。但主题数据是会不断更新的,不仅用户自己会创建主题,其他用户也会发表主题。那么如何为用户更新主页上呈现的主题数据呢?这里没有采取每次呈现都刷新的策略,而是在每次呈现主题表的第 1 页数据时才为用户重新装入主题数据。

接着修改主页,即 index.xhtml 文件。对主页的修改包括三个方面:

(1) 增加视图参数 p,并指定 preRenderView 事件的监听方法。

视图参数用于接收用户指定的当前页码,接收到的当前页码应保存到 currentPage 属性中。preRenderView 事件的监听方法应设置为受管 bean 的 initView 方法。

(2) 为 h:dataTable 标记添加 first 和 rows 属性。

(3) 在页面尾部添加产生页码超链接的代码。

代码清单 6-19 列出该页面文件所作的变动。其中,最后部分产生页码超链接的代码要用到 Facelets 标记库中 ui:repeat 标记。ui:repeat 标记可以对数组或表中的元素进行迭代处理和呈现。Facelets 标记库在本书第 10 章会有详细介绍。

代码清单 6-19　index.xhtml(部分)

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <f:metadata>
    <f:viewParam name="p" value="#{index.currentPage}"/>
    <f:event type="preRenderView" listener="#{index.initView}"/>
  </f:metadata>
  <h:head>
    <title>轻松论坛--主页</title>
    <h:outputStylesheet library="css" name="luntan_css.css"/>
  </h:head>
  <h:body>
    ……
    <h:form styleClass="margin0 width95">
      <h:dataTable value="#{index.topics}" var="topic" cellspacing="5px"
          first="#{index.pageSize * (index.currentPage-1)}" rows="#{index.pageSize}"
          styleClass="font1 width100"
          headerClass="text_left font2 bg_color">
        ……
```

```
            </h:dataTable>
        </h:form>

        <h:panelGroup layout="block" styleClass="margin10 text_center">
            <h:link value="&lt;" disabled="#{index.currentPage le 1}"
                outcome="index?p=#{index.currentPage-1}" includeViewParams="true"/>
             
            <h:panelGroup rendered="#{index.pages[0] gt 1}">
                <h:link value="1" outcome="index?p=1" includeViewParams="true"/>
                 
            </h:panelGroup>
            <h:panelGroup rendered="#{index.pages[0] gt 2}">
                <h:outputText value="..."/>
                 
            </h:panelGroup>
            <ui:repeat value="#{index.pages}" var="i">
                <h:link value="#{i}" disabled="true" style="background-color: silver"
                    includeViewParams="true" rendered="#{i==index.currentPage}"/>
                <h:link value="#{i}" outcome="index?p=#{i}" includeViewParams="true"
                    rendered="#{i!=index.currentPage}"/>
                 
            </ui:repeat>
            <h:panelGroup rendered="#{index.pages[index.pages.size()-1] lt index.pageCount-1}">
                <h:outputText value="..."/>
                 
            </h:panelGroup>
            <h:panelGroup rendered="#{index.pages[index.pages.size()-1] lt index.pageCount}">
                <h:link value="#{index.pageCount}" outcome="index?p=#{index.pageCount}"
                    includeViewParams="true"/>
                 
            </h:panelGroup>
            <h:link value="&gt;" outcome="index?p=#{index.currentPage+1}"
                disabled="#{index.currentPage ge index.pageCount}" includeViewParams="true"/>
        </h:panelGroup>

    </h:body>
</html>
```

在这里，实现翻页功能的页码超链接都是用结果超链接标记 h:link 生成的，所以单击它们都将产生一个 GET 请求。

6.5.3 修改“查看回复”页面

首先需要修改支撑该页面呈现的受管 bean 类，即 ShowReply.java。与修改 bean.Index 类相似，对该类的修改包括 3 个方面：

(1) 让该 bean 类扩展 util. Pager 类。

(2) 增加一个只读的 pageSize 属性。

(3) 修改 bean 类中原有的 initView() 方法,在其尾部添加设置 pageCount 和 showPages 属性以及调用 init 方法的代码。该方法是一个 preRenderView 事件监听方法。

代码清单 6-20 列出该类所作的变动。

代码清单 6-20　bean. ShowReply. java(部分)

```
public class ShowReply extends Pager{
  ......
  public void initView(){
    ......
    int pc=(int)Math.ceil(replys.size()*1.0/pageSize);
    setPageCount(pc);
    setShowPages(8);
    init();
  }
  ......
  private int pageSize=2;
  public int getPageSize(){
    return pageSize;
  }
}
```

最后需要修改"查看回复"页面本身,即 showReply. xhtml 文件。与修改 index. xhtml 文件类似,对该页面文件的修改也包括三个方面:

(1) 增加视图参数 p,用于接收用户指定的当前页码,接收到的当前页码应保存到 currentPage 属性中。

(2) 为 h:dataTable 标记添加 first 和 rows 属性。

(3) 在页面尾部添加产生页码超链接的代码。

由于相关修改代码与修改主页 index. xhtml 时的基本类似,这里不再列出。

6.6　小　　结

- 定义样式的一般格式:＜选择符＞{＜样式属性名＞:＜属性值＞;……},其中选择符可以是:标记选择符、类选择符、ID 选择符、属性选择符和伪类等。
- 使用样式或样式表的方式包括:定义内部样式表、定义内联样式、链接外部样式表、导入外部样式表、用 h:outputStylesheet 标记链接外部样式表。
- h:panelGrid 标记呈现为一个 HTML table 元素,主要用于页面布局。
- h:panelGroup 标记经常与 h:panelGrid 标记一起使用,用于将一些组件归组,以便它们被看作是一个组件放置在表格的一个单元格内。
- h:dataTable 标记呈现为一个 HTML table 元素,主要用于显示数据表格。其 value 属性可以指向一个数组或集合,标记会迭代处理(呈现)数组或集合的每个元素。其

var 属性用于暴露一个变量，该变量指向当前正在处理的元素。

- h:dataTable 标记中一般会嵌套若干个 h:column 子标记，用于配置数据表格中的列。
- f:facet 标记可以包含一个组件，其本身可嵌套于 h:dataTable 标记。通过对 f:facet 标记的 name 属性的设置，其子组件可被作为表格的标题、表头和表脚。

习 题 6

1. 简述使用 CSS 技术的优点。
2. 简述 h:outputStylesheet 标记的功能与用法。
3. 简述 h:panelGroup 标记的功能及其 layout 属性的用法。
4. 分别比较下面各组属性中各属性的作用与用法。

(1) columnClasses、rowClasses

(2) cellpadding、cellspacing

(3) captionClass、headerClass

(4) frame、rules

5. 利用 CSS 和 h:panelGrid 标记等技术，对第 4 章习题 5 的 sh4_logandreg 应用项目中的各页面进行布局和格式化，以达到满意的呈现效果。

6. 创建 JSF 应用 sh6_lookandbuy。该应用包括 3 个 JSF 页面、若干受管 bean 类和模型类，实现图书的检索、浏览与选购功能。请按下列步骤完成应用开发。

(1) 在“源包”中创建 4 个 Java 包：bean、entity、model 和 util。

(2) 在 entity 包中，创建一个表示图书的 Java 类 Book。下面是该类的不完整代码，请完善之。

```
public class Book {
    private String bookid;                  //图书编号(每本书的图书编号是唯一的)
    private String title;                   //书名
    private String author;                  //作者
    private String publisher;               //出版社
    private BigDecimal price;               //单价
    private String isbn;                    //ISBN 号
    private String banci;                   //版次
    private String yinci;                   //印次
    private String kaiben;                  //开本
    private String yeshu;                   //页数
    private String zishu;                   //字数
    private String zhuzhen;                 //装帧
    public Book() { }
    public Book(String bookid) {
        //初始化实例变量
    }
    public Book(String bookid,String title,String author,String publisher,
```

```
                BigDecimal price,String isbn){
        //初始化实例变量
    }
    /*
    * 为各实例变量提供相应的 getter 和 setter 方法
    */
    public boolean equals(Object object){
        //若参数是一个 Book 实例且其 bookid 值与当前图书的 bookid 相等,返回 true;
        //否则返回 false
    }
}
```

(3) 在 model 包中创建一个 DataBase 类,用于存放图书数据。下面是该类的不完整代码,请完善之。

```
public class DataBase {
    private static final List<Book>books=new ArrayList<Book>();
    static {
        //初始化 books: 添加 10 至 20 本图书
    }
    public static List<Book>getBooks(){
        return books;
    }
}
```

(4) 在 util 包中,创建一个与 JSF 应用 sh4_logandreg 中一样的 Java 类 ELUtil。然后再创建一个用于实现分页显示功能的 Java 类 Pager(参见第 6.5 节例子)。

(5) 在 model 包中创建一个表示选购项的 Java 类 Item。下面是该类的不完整代码,请完善之。

```
public class Item implements Serializable{
    private Book book;
    private int sl;                //数量
    /*
    * 为各实例变量提供相应的 getter 和 setter 方法
    */
    public void add(int n){
    //当前选购项的数量增加 n;
    }
}
```

(6) 在 model 包中创建一个实现图书管理业务的 BookManager 类。下面是该类的软件接口,请实现之。

```
public class BookManager {
    //根据图书编号返回相应的图书。如果不存在相应的图书,返回 null
    public Book findBookByBh(String bh);
```

```
    //返回所有图书
    public List<Book>queryBook();
    //查询书名中包含指定字符串的图书
    public List<Book>queryBook(String sm);
}
```

(7) 在 bean 包中创建一个受管 bean：名称为 sessionBean1、类名为 bean.SessionBean1、作用域为会话作用域。bean 类中定义一个可读写的 List<Item>型属性 cart。该属性的一个元素表示客户的一个选购项，其初值是一个空的 ArrayList<Item>表。

(8) 创建 JSF 页面 index.xhtml(主页)以及相应的类名为 bean.Index 的会话作用域受管 bean。页面的运行效果如图 6-7 所示。

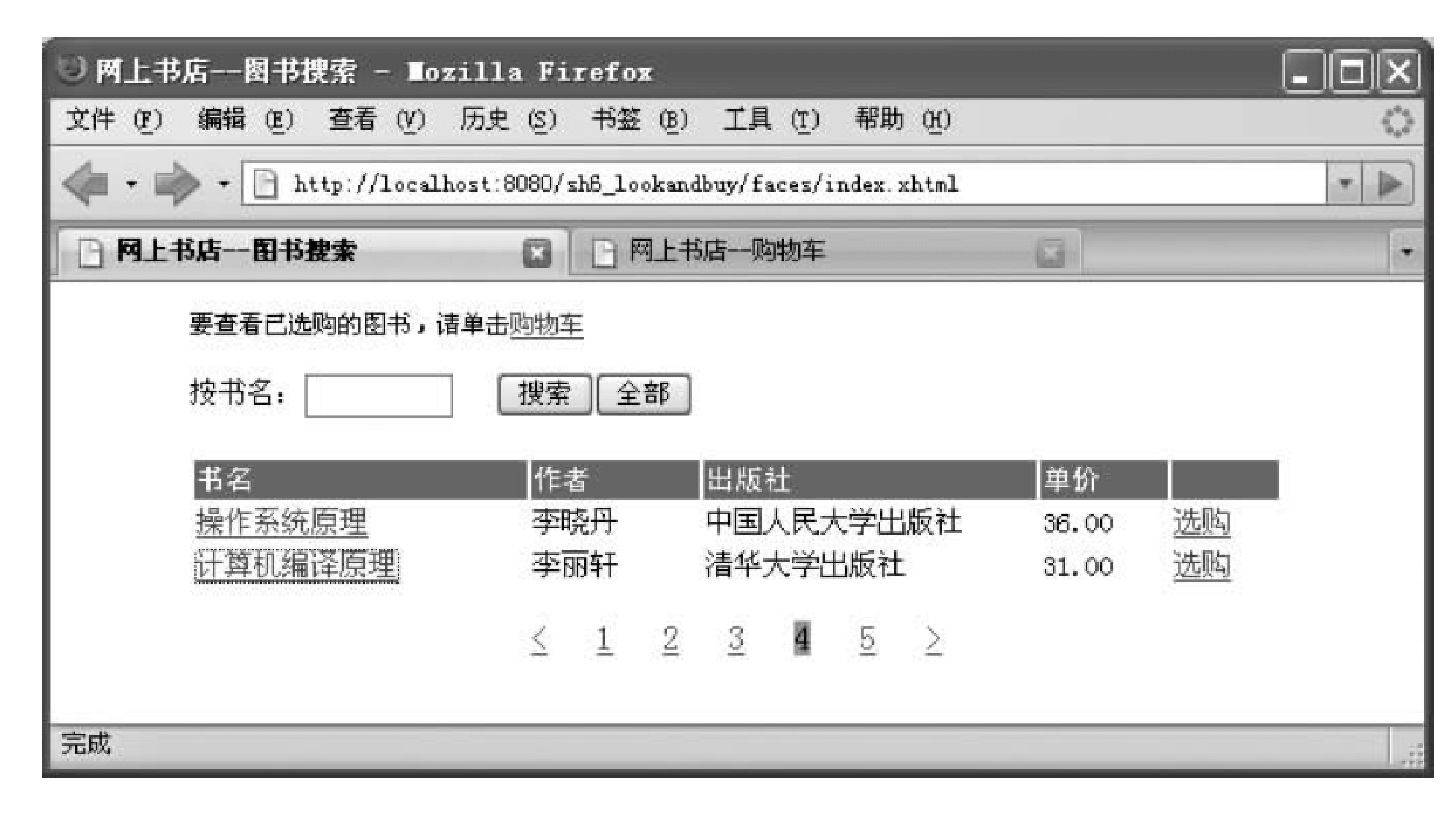


图 6-7　图书搜索页面(主页)

单击"搜索"按钮，页面将显示所有书名包含指定搜索串的图书信息。单击"全部"按钮，页面显示所有图书信息。

单击某图书书名的超链接，应用将导航至页面(bookdetail.xhtml)，显示该图书的详细信息。

单击"选购"按钮，应用将该图书添加至 sessionBean1 中的 cart 属性：如果购物车中已有该图书的选购项，则将其数量加 1；否则新添加一个选购项，并将其数量设置为 1。然后重新显示本页面。

该页面分页显示图书信息。页面中应提供相应的页码超链接以实现翻页功能。

单击"购物车"超链接可导航至购物车页面，购物车页面在名为"cart"的新窗口或标签页中打开。

(9) 创建页面 bookdetail.xhtml 以及相应的名为 bean.Bookdetail 的会话作用域受管 bean。页面的运行效果如图 6-8 所示。

单击"添加至购物车"按钮时，应用将该图书添加至 sessionBean1 中的 cart 属性：如果购物车中已有该图书的选购项，则将其数量加 1；否则新添加一个选购项，并将其数量设置为 1。然后导航至购物车页面，购物车页面在名为"cart"的新窗口或标签页中打开。



图 6-8　图书详细信息页面

（10）创建页面 cart.xhtml 以及相应的名为 bean.Cart 的会话作用域受管 bean。页面的运行效果如图 6-9 所示。

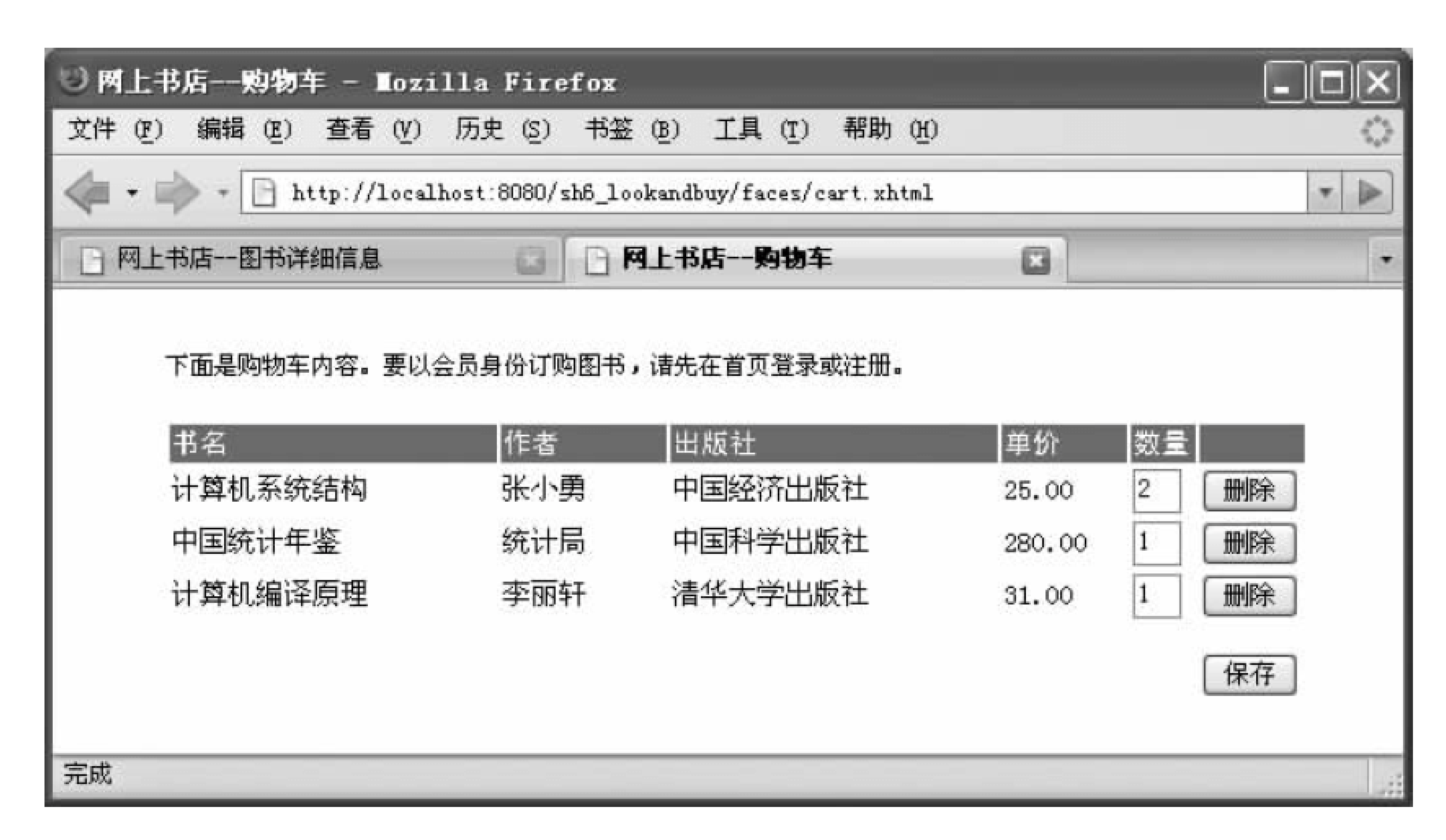


图 6-9　购物车页面

单击“删除”按钮，可以从购物车中删除相应的选购项；单击“保存”按钮，可以保存用户对各选购项所做的数量修改。

第 7 章　转换器与验证器

本章主题：

- 标准转换器
- 注册转换器类
- 引用转换器
- 自定义转换器
- 标准验证器
- 注册验证器类
- 引用验证器
- 自定义验证器

在 JSF 应用中，来自客户端的请求参数被作为组件的值，期间会经历接收、转换和验证等过程。接收发生于请求处理生命周期的“应用请求值”阶段，指每个组件读取属于自己的请求参数，称为被提交值。转换和验证通常发生于请求处理生命周期的“处理验证”阶段。对于每个组件，首先会对被提交值进行转换处理，若转换成功，则继续进行验证处理。如果转换或验证出错，可以抛出相应的例外，例外中通常包含一个 FacesMessage 实例作为出错信息。如果所有组件的转换和验证都获得通过，则进入“更新模型值”阶段；否则直接进入“呈现响应”阶段，转换和验证过程中产生的出错信息会呈现在响应页面中。如果组件的 immediate 属性值为 true，那么其转换和验证处理会在应用请求值阶段立即进行。

本章介绍 JSF 转换模型和验证模型以及相关的编程技术。

7.1　转换器概述

转换器服务于组件，用于完成服务器端的组件值与客户端的组件值表示之间的转换，包括输入转换和输出转换。服务器端的组件值可以是各种 Java 类型数据（对象），客户端的组件值表示则通常是字符串文本。转换器的输入转换功能是将一定格式的请求参数（字符串文本）转换成特定类型的 Java 对象，发生于“处理验证”阶段。转换器的输出转换功能是将特定类型的 Java 对象转换成一定格式的字符串文本以供显示，发生于“呈现响应”阶段。

对于各种转换器，输入转换的源与输出转换的目标都是字符串文本。对于某种转换器，输入转换的目标与输出转换的源总是同一种 Java 类型对象。通常也把这种 Java 数据类型称作是该种转换器能够支持转换的数据类型。

能接受转换器服务的组件应该是实现 ValueHolder 接口的组件，包括输入类组件、输出类组件、视图参数组件、结果类组件等。

转换器模型的编程主要涉及定义转换器类、注册转换器类和引用转换器等三方面的工作。

1. 定义转换器类

转换器类是实现 Converter 接口的类，主要包括两个实例方法，分别提供输入转换和输出转换的功能。

JSF 框架本身提供一组所谓的标准转换器，可以支持一些常见的 Java 数据类型的转换。

2. 注册转换器类

一般来说，要使用转换器，需要先向 JSF 应用注册相应的转换器类。通过注册转换器类，JSF 框架或页面制作者可以更好地为组件引用所需的转换器。

JSF 框架提供的各标准转换器类在 JSF 应用启动时都自动进行了注册。

3. 引用转换器

引用转换器是指在组件上注册指定类型的转换器对象，以便为该组件提供相应的转换服务。调用 ValueHolder 型组件的 setConverter 方法可以为该组件注册指定的转换器。更多情况下，可以通过声明的方式为组件注册指定类型的转换器。

7.2 使用标准转换器

本节介绍 JSF 框架自含的标准转换器，包括标准转换器的种类、注册情况、引用方法以及标准转换错误消息等。

7.2.1 标准转换器简介

标准转换器由 JSF 规范定义、随 JSF 实现一起提供给用户，主要用于一些常见的 Java 类型的组件值与其字符串文本表示之间的转换。表 7-1 给出了各标准转换器类的类名及其支持转换的 Java 数据类型，其中标准转换器类都属于 javax. faces. convert 包，其支持转换的数据类型无特殊指明都属于 java. lang 包。

表 7-1 标准转换器及其支持转换的数据类型

标准转换器类	支持转换的数据类型	标准转换器类	支持转换的数据类型
ByteConverter	Byte	BooleanConverter	Boolean
ShortConverter	Short	BigDecimalConverter	java. math. BigDecimal
IntegerConverter	Integer	BigIntegerConverter	java. math. BigInteger
LongConverter	Long	EnumConverter	Enum(枚举类型)
CharacterConverter	Character	DateTimeConverter	java. util. Date
FloatConverter	Float	NumberConverter	Number
DoubleConverter	Double		

要使用转换器，通常需先向 JSF 应用注册相应的转换器类。转换器类可按 ID(标识符)注册，也可按类型(该转换器支持转换的数据类型)注册，或者既按 ID 注册，又按类型注册。按 ID 注册后，开发人员就可以在页面文件中通过 ID 为某组件指定其所需要的转换器。按类型注册后，当某组件需要进行这种类型的转换时，JSF 框架会自动创建对应类型的转换器

对象，提供相应的转换服务。

当 JSF 应用启动，所有的标准转换器都会自动完成按 ID 注册或者按类型注册，如表 7-2 所示。其中，转换器 EnumConverter 只按类型进行了注册，转换器 DateTimeConverter 和转换器 NumberConverter 都只按 ID 进行了注册。

表 7-2 标准转换器的注册

标准转换器类	注册 ID	是否按类型注册
ByteConverter	javax. faces. Byte	√
ShortConverter	javax. faces. Short	√
IntegerConverter	javax. faces. Integer	√
LongConverter	javax. faces. Long	√
CharacterConverter	javax. faces. Character	√
FloatConverter	javax. faces. Float	√
DoubleConverter	javax. faces. Double	√
BooleanConverter	javax. faces. Boolean	√
BigDecimalConverter	javax. faces. BigDecimal	√
BigIntegerConverter	javax. faces. BigInteger	√
EnumConverter	无	√
DateTimeConverter	javax. faces. DateTime	否
NumberConverter	javax. faces. Number	否

标准转换器的这种注册应该被看做是一种默认注册。开发人员可以用相同的 ID 或类型名注册自定义的转换器类，这样就会覆盖这种默认注册。

7.2.2 引用转换器

引用转换器是指为特定组件指定所需的转换器，以便为组件值的转换提供服务。引用转换器的方式有多种，下面分别介绍。

1. 默认方式

如果组件标记的 value 属性与受管 bean 的某个属性关联，那么 JSF 框架会为该组件自动创建一个支持该 bean 属性类型转换的转换器对象。此种方式要求相应的转换器类已按类型进行了注册。

例如，下面 h:inputText 标记的 value 属性指向 myBean 受管 bean 的 num 属性：

```
<h:inputText value="#{myBean.num}" />
```

如果 num 属性的类型为 java. lang. Integer，那么在默认情况下(还没有为该类型注册自定义的转换器类)，JSF 框架会为该标记组件自动创建 java. faces. convert. IntegerConverter 型转换器，提供从字符串文本到 Integer 型数据和从 Integer 型数据到字符串文本的转换服务。

又例如，下面 h:outputText 标记的 value 属性指向 myBean 受管 bean 的 sex 属性：

```
<h:outputText value="#{myBean.sex}" />
```

如果 sex 属性的类型为 boolean(java. lang. Boolean)，那么在默认情况下，JSF 框架会为该标记组件自动创建 java. faces. convert. BooleanConverter 型转换器，提供从 Boolean 型数据到字符串文本的转换服务。

2. 通过 ID 引用转换器

将组件标记的 converter 属性或 f:converter 子标记的 converterId 属性设置为某转换器类的注册 ID，此时，JSF 框架会为该标记组件创建指定类型的转换器对象。此种方式要求转换器类已按 ID 进行了注册。

下面是使用该种方式引用转换器的示例。

```
<h:inputText converter="javax.faces.Integer"/>
```

或：

```
<h:inputText>
    <f:converter converterId="javax.faces.Integer"/>
</h:inputText>
```

这里，javax. faces. Integer 是 javax. faces. convert. IntegerConverter 转换器类的注册 ID。JSF 框架会在需要时为组件自动创建该种类型的转换器对象，提供相应的转换服务。

3. 直接引用转换器

让组件标记的 f:converter 子标记的 binding 属性指向某受管 bean 的一个属性，而该属性的值是一个转换器对象。此种方式不要求相关的转换器类已经注册。

下面是使用该种方式引用转换器的示例，其中 f:converter 子标记的 binding 属性通过值表达式指向 myBean 受管 bean 的 cvt 属性。

```
<h:inputText>
    <f:converter binding="#{myBean.cvt}"/>
</h:inputText>
```

myBean 受管 bean 的 cvt 属性的类型应该是一个转换器类，并能返回该种类型的一个转换器对象。例如，下面代码定义了一个可读写的 cvt 属性，可返回一个 IntegerConverter 型转换器。

```
IntegerConverter cvt=new IntegerConverter();
public IntegerConverter getCvt(){
    return cvt;
}
public void setCvt(IntegerConverter cvt){
    this.cvt=cvt;
}
```

上面介绍的引用转换器的方法不仅适用于标准转换器，也适用于后面要介绍的自定义转换器。对标准的 DateTimeConverter 转换器和 NumberConverter 转换器，JSF 还提供用

于引用转换器的专用标记，下面分别介绍。

7.2.3 DateTimeConverter 转换器

日期时间转换器(DateTimeConverter)是一种标准转换器。默认情况下，这种转换器只按 ID 进行了注册，而没有按类型进行注册。

虽然可以采用上述方式 2 或方式 3 引用日期时间转换器，但更多的情况下会采用这种转换器的专用标记 f:convertDateTime。采用 f:convertDateTime 标记引用日期时间转换器时，可以通过设置标记的属性来控制转换器的转换行为。

下面介绍该专用标记的属性及其用法。

(1) type：指定日期时间字符串包含的内容，取值范围 date、time 和 both，默认值为 date。

(2) dateStyle：指定日期部分的样式，仅在设置了 type 属性值为“date”或“both”时有效。取值范围包括：

- short：09-10-23、10/23/09。
- medium(默认)：2009-10-23、Oct 23，2009。
- long：2009 年 10 月 23 日、October 23，2009。
- full：2009 年 10 月 23 日 星期五、Friday，October 23，2009。

说明：上面每一个取值都有两个不同的示例，这取决于 locale 属性的设置。对相同的日期样式，如果设置了不同的场所、语言，那么会有不同的表示形式。

(3) timeStyle：指定时间部分的样式，仅在设置了 type 属性值为“time”和“both”时有效。取值范围包括：

- short：3:25 PM、下午 3:25。
- medium(默认)：3:25:50 PM、15:25:50。
- long：3:25:50 PM CST、下午 03 时 25 分 50 秒。
- full：3:25:50 PM CST、下午 03 时 25 分 50 秒 CST。

说明：上面每一个取值都有两个不同的示例，这取决于 locale 属性的设置。对相同的时间样式，如果设置了不同的场所、语言，那么会有不同的表示形式。

(4) locale：指定场所。一个 Locale 对象或表示语言代码的字符串，如 en、zh。一般来说，一个场所的语言特性会影响日期时间数据的具体表示形式。默认情况下，该属性值为本地场所。

(5) timeZone：指定时区。一个表示时区 ID 的字符串，如 PRC、GMT＋8。实际上，Date 对象存放的是格林尼治时间，所以在默认情况下，该转换器接收或产生的字符串文本都被看作是格林尼治时间。如果要使用其他时区的时间，应该设置该属性。

(6) pattern：直接指定日期时间的字符串文本表示的模式。指定该属性后，type、dateStyle、timeStyle 都将失效。在指定模式时可使用下面专用字符(区分大小写)：

- M：月份，如 MM(08)、MMM(八月)。
- d：日期，如 dd(05)。
- y：年份，如 yy(12)、yyyy(2012)。
- E：星期几，如 EEE(星期六)。
- H：小时(0-23)。

- K：小时(0-11)。
- m：分，如 mm(15)。
- s：秒，ss(45)。
- S：毫秒，如 SSS(865)。
- a：上、下午标记，如 a(下午)。

下面标记中，假设 createTime 属性是 Date 型的。引用转换器标记指定既显示日期也显示时间，时区为北京(GMT+8)。

```
<h:outputText value="#{myBean.createTime}">
    <f:convertDateTime type="both" timeZone="PRC"/>
</h:outputText>
```

如果本地场所为 Locale.CHINA，那么上面标记的呈现效果类似如下：

```
2012-6-27 13:22:01
```

如果在引用转换器标记中添加属性设置：timeStyle="long"，那么标记的呈现效果类似如下：

```
2012-6-27 下午 01 时 22 分 01 秒
```

下面标记中，假设 expirationDate 属性是 Date 的。引用转换器标记通过 pattern 属性指定了日期时间的字符串文本表示的模式。

```
<h:inputText value="#{myBean.expirationDate}">
    <f:convertDateTime pattern="MM/yyyy" timeZone="PRC"/>
</h:inputText>
```

呈现时，文本域中仅显示 expirationDate 属性值的月份与年份，两者用/分隔。输入时，也应该按该模式输入，否则转换器将无法理解而出错或出现错误结果。由于仅指定了年份和月份，输入结果的日期将被设置为 1、时间则被设置为 0。

在论坛应用中，主题表和回复表中都包含 Date 型数据的显示，如主题的创建时间、最后回复时间等。为了能按所需的格式显示这些时间，应用专门编写了一个实用方法 getTime(Date)。若一个页面要显示时间，可以先调用该方法将 Date 型数据转换成一定格式的字符串，然后再显示。例如，主题表中显示创建时间的标记：

```
<h:outputText value="#{index.showTime(topic.createtime)}" styleClass="font3"/>
```

其中，受管 bean 的 showTime 方法会调用实用方法 getTime(Date)将主题的创建时间转换成一定格式的字符串。

有了这个标准的日期时间转换器后，就不需要这么麻烦了。可以直接引用该转换器，将一个 Date 型数据转换成所需格式的字符串输出。例如，主题表中显示创建时间的标记可以改写成如下：

```
<h:outputText value="#{topic.createtime}" styleClass="font3">
    <f:convertDateTime type="both" dateStyle="medium" timeStyle="short"/>
</h:outputText>
```

7.2.4 NumberConverter 转换器

数值转换器(NumberConverter)是一种标准转换器。默认情况下,这种转换器只按 ID 进行了注册,而没有按类型进行注册。

java.lang.Number 是一个表示数值的抽象超类,其子类包括:

- java.lang.Byte。
- java.lang.Short。
- java.lang.Integer。
- java.lang.Long。
- java.lang.Float。
- java.lang.Double。
- java.math.BigInteger。
- java.math.BigDecimal。

NumberConverter 转换器支持上述各种类型的组件值与字符串文本之间的转换。对于与组件值绑定在一起的受管 bean 属性,其类型也可以是相应的基本类型,如 int、double 等。

与日期时间转换器一样,数值转换器也有其专用的引用标记 f:convertNumber。采用 f:convertDateTime 标记引用日期时间转换器时,可以通过设置标记的属性来控制转换器的转换行为。

下面是该专用标记的一些属性及其用法。

(1) type:指定表示数值的字符串文本的类型,取值范围包括:

- number(默认值):普通数值形式。
- currency:货币形式。
- percent:百分数形式。

(2) minIntegerDigits:指定输出产生的字符串文本中整数部分的最少位数,不足时高位以 0 填充。对输入无影响。

(3) maxIntegerDigits:指定输出产生的字符串文本中整数部分的最多位数,超出时截去高位。对输入无影响。

(4) minFractionDigits:指定输出产生的字符串文本中小数部分的最少位数,不足时低位以 0 填充。对输入无影响。

(5) maxFractionDigits:指定输出产生的字符串文本中小数部分的最多位数,超出时低位采用 RoundingMode.HALF_EVEN 模式舍入。对输入无影响。

(6) integerOnly:指定在输入转换时是否只解析整数部分,默认值为 false。对输出无影响。

(7) locale:指定场所,一个 Locale 对象或表示语言代码的字符串。一般来说,如果没有其他属性的特别指定,货币形式输出时的货币符号会由该属性指定的场所的国家特性决定。默认情况下,该属性值为本地场所。

(8) currencySymbol:指定货币符号,仅当 type 属性为“currency”时有效。

(9) currencyCode:指定使用 ISO 定义的货币代码,如 CNY、HKD、USD 等,仅当 type 属性为“currency”时有效。

(10) groupingUsed:指定是否使用分组符号。默认情况下,使用分组符号。只影响输出。

(11) pattern：指定数值的字符串文本表示的模式。具体写法可参考 DecimalFormat 类的 API 文档。

假设 myBean 受管 bean 中定义有以下可读属性：

```
private double value=-12586.346;
public double getValue(){
  return value;
}
```

下面组件标记用于输出该 bean 属性值，其中引用数值转换器标记指定采用货币形式输出。

```
<h:outputText value="#{my.value}">
    <f:convertNumber type="currency"/>
</h:outputText>
```

如果本地场所为 Locale.CHINA，那么上面标记的呈现效果应该为：

```
-￥12,586.35
```

将引用数值转换器标记改成如下，显式指明货币代码为 USD。

```
<f:convertNumber type="currency" currencyCode="USD"/>
```

那么不管本地场所为何值，上面组件标记的输出都应该是：

```
-USD12,586.35
```

将引用数值转换器的标记改成如下，最多小数位数和最少小数位数都设置为 4。

```
<f:convertNumber maxFractionDigits="4" minFractionDigits="4"/>
```

那么上面组件标记的呈现效果应该是：

```
-12,586.3460
```

将引用数值转换器的标记改成如下，其中用 pattern 属性指定了字符串文本的模式。

```
<f:convertNumber pattern="#,###.0000"/>
```

模式指定整数部分采用分组符号(,)，每 3 位为一组；小数部分为 4 位。这种情况的输出效果与第三种情况是一样的，即：

```
-12,586.3460
```

7.2.5 转换错误

在请求处理生命周期的“处理验证”阶段，当一个组件的输入转换处理抛出例外时，通常会产生一个错误消息(FacesMessage 对象)放入消息队列，之后其他组件的转换和验证处理还会继续进行。当该阶段结束时，如果之前曾有例外抛出，那么就会跳过其他阶段，直接进入“呈现响应”阶段。呈现的页面中会包含那些无法完成转换的输入值，用户可以重新修改输入。另外，页面文件中通常应包含相应的 h:message 或 h:messages 组件标记，以便把转

换过程中产生错误消息呈现出来。

在“呈现响应”阶段，如果一个组件的输出转换处理抛出例外，JSF 框架将会产生错误页面作为响应。

JSF 框架提供一组标准转换错误消息资源，每个消息资源包含一个 ID 和一个默认的消息文本，如表 7-3 所示。标准转换器在转换处理中发生错误时，都会以其中某个消息资源文本为模板产生错误消息，消息文本中的参数会用相应的具体内容代替。

表 7-3　标准转换错误消息

消息 ID	消 息 文 本
javax.faces.converter.IntegerConverter.INTEGER	{2}: "{0}" must be a number consisting of one or more digits.
javax.faces.converter.IntegerConverter.INTEGER_detail	{2}: "{0}" must be a number between -2147483648 and 2147483647. *Example: {1}*
javax.faces.converter.DoubleConverter.DOUBLE	{2}: "{0}" must be a number consisting of one or more digits.
javax.faces.converter.DoubleConverter.DOUBLE_detail	{2}: "{0}" must be a number between 4.9E-324 and 1.7976931348623157E308. *Example: {1}*
* javax.faces.converter.BooleanConverter.BOOLEAN_detail	{1}: "{0}" must be 'true' or 'false'. *Any value other than 'true' will evaluate to 'false'.*
javax.faces.converter.BigDecimalConverter.BIGINTEGER_detail	"{0}" must be a number consisting of one or more digits. *Example: {1}*
javax.faces.converter.BigDecimalConverter.BIGDECIMAL_detail	"{0}" must be a signed decimal number *consisting of zero or more digits, that may be followed by a decimal point and fraction. Example: {1}*
javax.faces.converter.NumberConverter.NUMBER_detail	{2}: "{0}" is not a number. *Example: {1}*
javax.faces.converter.NumberConverter.CURRENCY_detail	{2}: "{0}" could not be understood as a currency value. *Example: {1}*
javax.faces.converter.NumberConverter.PERCENT_detail	{2}: "{0}" could not be understood as a percentage. *Example: {1}*
javax.faces.converter.DateTimeConverter.DATE_detail	{2}: "{0}" could not be understood as a date. *Example: {1}*
javax.faces.converter.DateTimeConverter.TIME_detail	{2}: "{0}" could not be understood as a time. *Example: {1}*
javax.faces.converter.DateTimeConverter.DATETIME_detail	{2}: "{0}" could not be understood as a date and time. *Example: {1}*
javax.faces.converter.EnumConverter.ENUM	{2}: "{0}" must be convertible to an enum.
javax.faces.converter.EnumConverter.ENUM_detail	{2}: "{0}" must be convertible to an enum from the enum that contains the constant "{1}".
* javax.faces.converter.EnumConverter.ENUM_NO_CLASS_detail	{1}: "{0}" must be convertible to an enum from the enum, but no enum class provided.
* javax.faces.converter.STRING	{1}: Could not convert "{0}" to a string.

说明：

(1) 当概要文本是详细文本的子串时，只列出详细文本，其中斜体部分是概要文本不具有的。

(2) 对大多数消息文本，{0}为转换失败的提交值，{1}为有效值示例，{2}是组件标签。对消息 ID 前打 * 的消息文本，{0}为转换失败的提交值，{1}是组件标签。

(3) 表中最后一个消息文本是各标准转换器在输出转换出错时要用到的，其他各消息文本都是某标准转换器在输入转换出错时要用到的。

如果对组件的 converterMessage 属性进行了设置，那么该属性值将代替由相关转换器或独立转换方法产生的错误消息显示于响应页面。

也可以通过自定义消息包来覆盖上述标准转换错误消息文本，即让标准转换器基于开发人员指定的消息文本产生转换错误消息。该技术将在第 9.2 节详细介绍。

7.3 自定义转换器

标准转换器能满足大多数一般性的转换需求，但一些特殊的转换任务还需要自定义转换器来完成。本节介绍自定义转换器类的编写及注册。

7.3.1 编写自定义转换器类

转换器类必须实现转换器接口，即 javax.faces.convert.Converter 接口。Converter 接口声明了输入转换方法 getAsObject 和输出转换方法 getAsString，分别对应转换器的两种功能。

输入转换方法用于将指定的字符串文本转换成特定类型的数据，下面是其方法签名。

```
Object getAsObject(FacesContext context, UIComponent component, String value)
                        throws ConverterException
```

方法包含三个参数，当 JSF 框架要调用该方法对一个组件值进行输入转换时，会自动传入这些参数。

- context：当前请求上下文，保存了与当前请求处理相关的所有信息。
- component：正被转换的 UI 组件。
- value：要被转换的字符串文本。

如果转换失败，方法可以抛出一个不受检查的运行时 ConverterException 例外。该例外在创建时通常应携带一个 FacesMessage 消息，JSF 框架在处理例外时，会将该消息压入消息队列。当所在阶段结束时，控制将直接进入呈现响应阶段，相应的消息可以被显示在页面上。

输出转换方法用于将指定的特定类型数据转换成字符串文本，下面是其方法签名。

```
String getAsString(FacesContext context, UIComponent component, Object value)
                        throws ConverterException
```

方法包含三个参数，当 JSF 框架要调用该方法对一个组件值进行输出转换时，会自动传入这些参数。

- context：当前请求上下文，保存了与当前请求处理相关的所有信息。
- component：正被转换的 UI 组件。
- value：要被转换的组件值。

输出转换方法通常不应出现错误，但一旦出现转换错误，方法也可以抛出一个不受检查的运行时 ConverterException 例外。此时，JSF 框架将呈现一个错误页面。

7.3.2 注册自定义转换器类

转换器在使用之前通常应先注册其转换器类。转换器类可按 ID 注册，也可按类型注册，或者既按 ID 注册又按类型注册。

前面已经介绍，标准转换器类在 JSF 应用启动时会自动完成相应的注册，但自定义转换器类则需要开发人员自行注册。注册转换器类的方法有两种：一是在 Faces 配置文件中用 converter 元素进行声明；二是用@FacesConverter 型标注修饰转换器类。

下面是在 Faces 配置文件中按类型注册转换器类的一个示例。

```
<faces-config ......>
    <converter>
        <converter-for-class>java.lang.Integer</converter-for-class>
        <converter-class>converter.MyIntegerConverter</converter-class>
    </converter>
</faces-config>
```

其中，converter-for-class 子元素指定该种转换器所支持的转换类型，converter-class 子元素指定该转换器类的完整类名。

下面是采用 Java 标注方式按类型注册转换器类的一个示例。

```
@FacesConverter(forClass=java.lang.Integer.class)
public class MyInteger implements Converter {...}
```

其中，@FacesConverter 标注的 forClass 属性指定该转换器支持转换的数据类型。

下面是在 Faces 配置文件中按 ID 注册转换器类的一个示例。

```
<faces-config ......>
    <converter>
        <converter-id>myconverter</converter-id>
        <converter-class>converter.MyIntegerConverter</converter-class>
    </converter>
</faces-config>
```

其中，converter-id 子元素指定注册的转换器类的注册 ID。

下面是采用 Java 标注方式按 ID 注册转换器类的一个示例。此时只需要在标注中直接指定转换器类的注册 ID。

```
@FacesConverter("myconverter")
public class MyInteger implements Converter {...}
```

需要时，可以用自定义转换器替换标准转换器。例如在完成上述自定义转换器类的注

册后，如果采用默认方式引用转换器，那么 JSF 框架会使用自定义转换器完成 Integer 型与字符串文本之间的转换；如果通过指定注册 ID 方式引用转换器，那么当指定注册 ID 为 myconverter 时，将使用自定义转换器完成 Integer 型与字符串文本之间的转换，当指定注册 ID 为 javax.faces.Integer 时，将使用标准转换器完成相应的转换。

7.3.3 自定义转换器应用示例

该应用项目(ch7_demo)主要由两个页面、一个受管 bean 和一个自定义转换器类组成。项目的运行效果如图 7-1 所示。



图 7-1 应用 ch7_demo 运行效果图

该应用的主要功能是利用一个自定义转换器对用户输入的 ISBN 图书编号进行格式转换。ISBN 编号由一串数字组成，这串数字可分为几组，每组之间可以不做分隔，也可以用空格或减号(—)分隔。应用允许用户采用任何一种格式输入，但经转换器处理将产生统一的不含分隔符、完全由数字组成的 ISBN 编号。

如果用户输入的 ISBN 编号成功通过转换，应用将导航至另一个页面显示该 ISBN 编号；否则呈现原来的页面，并显示转换错误消息。

1. 自定义转换器

自定义转换器类 ISBNConverter 定义于 converter 包，见代码清单 7-1。输入转换的主要功能是去除输入的 ISBN 编号中空白符号(包括空格)和减号(—)、返回完全由数字组成 ISBN 编号。如果输入的 ISBN 编号中包含其他字符，则产生转换出错消息、并抛出例外。输出转换只是返回 ISBN 编号本身，没有做额外的处理。

代码清单 7-1　自定义转换器类(converter. ISBNConerter. java)

```java
package converter;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;

public class ISBNConverter implements Converter{

  public Object getAsObject(FacesContext context,UIComponent component,String value)
                           throws ConverterException {
    StringBuilder sb=new StringBuilder(value);
    int i=0;
    while(i<sb.length()){
      char ch=sb.charAt(i);
      if(Character.isDigit(ch)){
        i++;
      } else if(Character.isWhitespace(ch)||ch=='-'){
        sb.deleteCharAt(i);
      } else {
        String s="ISBN 中出现非法字符:"+ch;
        FacesMessage msg=new FacesMessage(s,s);
        throw new ConverterException(msg);
      }
    }
    return sb.toString();
  }

  public String getAsString(FacesContext context,UIComponent component,Object value){
    return String.valueOf(value);
  }
}
```

该自定义转换器类按 ID 进行注册，在 Faces 配置文件(faces-config. xml)中进行声明，见代码清单 7-2。

代码清单 7-2　注册自定义转换器类(converter. ISBNConverter)

```xml
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.0" ......>
  <converter>
    <converter-id>isbnconverter</converter-id>
    <converter-class>converter.ISBNConverter</converter-class>
  </converter>
  ......
</faces-config>
```

2. 受管 bean

应用的受管 bean 是请求作用域的，其中定义了一个可读写的 isbn 属性，具体代码如代码清单 7-3 所示。

代码清单 7-3　受管 bean(MyBean.java)

```
package bean;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class MyBean {

  private String isbn;
  public String getIsbn(){
    return isbn;
  }
  public void setIsbn(String isbn){
    this.isbn=isbn;
  }
}
```

3. JSF 页面

该应用包含两个页面。index.xhtml(代码清单 7-4)是一个欢迎页面，提供一个表单，允许用户输入一个 ISBN 编号。输入的 ISBN 编号经自定义转换器转换后保存在受管 bean 的 isbn 属性中。

代码清单 7-4　index.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>ConverterDemo</title>
  </h:head>
  <h:body>
    <h:form>
      <h:outputLabel for="isbn" value="ISBN:"/>
      <h:inputText id="isbn" value="#{myBean.isbn}">
        <f:converter converterId="isbnconverter"/>
      </h:inputText>
      <h:commandButton value="OK" action="response"/>
      <p/>
```

```
            <h:message for="isbn"/>
        </h:form>
    </h:body>
</html>
```

response. xhtml(代码清单 7-5)是一个响应页面,只是简单地将保存在受管 bean 中的 isbn 属性值读出显示。

代码清单 7-5　response. xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>ConverterDemo</title>
    </h:head>
    <h:body>
        <h:outputLabel for="isbn" value="ISBN:"/>
        <h:outputText id="isbn" value="#{myBean.isbn}"/>
    </h:body>
</html>
```

7.4　验证器概述

验证器服务于组件,用于对已经经过输入转换的组件的被提交值进行有效性验证,如数值型的组件值是否在指定的范围内、String 型的组件值的长度是否小于约定的值等。验证服务发生于请求处理生命周期的“处理验证”阶段。在该阶段,先对组件的被提交值进行输入转换,如果转换成功,则接着进行验证处理。如果验证成功,则产生组件的本地值,若该值不同于组件原来的值,则引发组件的值变化事件,并被压入事件队列。

能接受验证器服务的组件应该是实现 EditableValueHolder 接口的组件,包括输入类组件、视图参数组件等。

验证器模型的编程主要涉及定义验证器类、注册验证器类和引用验证器等三方面的工作。

1. 定义验证器类

验证器类是实现 Validator 接口的类,主要包含一个实例方法,提供对某组件值的特定的验证处理功能。

JSF 框架本身提供一组所谓的标准验证器,可以完成一些基本的组件值验证处理功能。

2. 注册验证器类

一般来说,要使用验证器,需要先向 JSF 应用注册相应的验证器类。通过注册验证器类,页面制作者可以更好地为组件引用所需的验证器。

JSF 框架提供的各标准验证器类在 JSF 应用启动时都自动进行了注册。

3. 引用验证器

引用验证器是指在组件上注册指定类型的验证器对象，以便为该组件值提供相应的验证服务。调用 EditableValueHolder 型组件的 setValidator 方法可以为该组件注册指定的验证器。更多情况下，可以通过声明的方式为组件注册指定类型的验证器。

一个组件可以注册多个验证器对象，这些验证器的验证功能会被依次调用。

7.5 使用标准验证器

本节介绍 JSF 框架自含的标准验证器，包括标准转换器的种类、注册情况、引用方法以及标准验证错误消息等。

7.5.1 标准验证器简介

标准验证器由 JSF 规范定义、随 JSF 实现一起提供给用户，主要用于完成一些基本的组件值验证处理功能。表 7-4 给出了几个主要的标准验证器类的类名及其功能说明，其中标准验证器类都属于 javax. faces. validator 包。

表 7-4 标准验证器

验证器类名	功 能 说 明
DoubleRangeValidator	检测数值型的组件值是否在指定的范围内
LengthValidator	检测 String 型的组件值长度是否在指定范围内
LongRangeValidator	检测整型组件值是否在指定范围内
RegexValidator	检测组件值是否匹配指定的正则表达式
RequiredValidator	强制组件值不能为空。相当于将组件标记的 required 属性设置为 true

各标准验证器类在 JSF 应用启动时都自动进行了注册。与转换器类不同，验证器类只按 ID(标识符)进行注册。注册了验证器类，页面制作者就可以通过指定注册 ID 来引用指定类型的验证器。对标准验证器，更一般的引用方式是使用其专用的 JSF 标记。表 7-5 列出了各标准验证器类的注册 ID 以及引用该验证器的专用 JSF 标记。

表 7-5 标准验证器的注册 ID 与引用标记

验 证 器 类	注册 ID	引 用 标 记
DoubleRangeValidator	javax. faces. DoubleRange	f:validateDoubleRange
LengthValidator	javax. faces. Length	f:validateLength
LongRangeValidator	javax. faces. LongRange	f:validateLongRange
RegexValidator	javax. faces. RegularExpression	f:validateRegex
RequiredValidator	javax. faces. Required	f:validateRequired

这里，前三个标准验证器的引用标记都包含 minimum 和 maximun 属性，用以指定允许

值的下限和上限;RegexValidator 验证器的引用标记包含 pattern 属性,用以指定一个正则表达式。

7.5.2 引用验证器

引用验证器是指为特定组件指定所需的验证器,以便为组件提供值验证服务。引用验证器的方式有多种,下面分别介绍。

1. 通过 ID 引用验证器

将组件标记的 f:validator 子标记的 validatorId 属性设置为某验证器类的注册 ID,此时,JSF 框架会为该组件创建指定类型的验证器对象。此种方式要求验证器类已按 ID 进行了注册。下面是使用该种方式引用验证器的示例。

```
<h:inputText>
    <f:validator validatorId=" javax.faces.Required"/>
</h:inputText>
```

2. 引用独立的验证方法

可以通过组件标记的 validator 属性注册一个独立的验证方法。下面代码演示了这种方法的使用。

```
<h:inputText value="#{myBean.value}" validator="#{myBean.validate}/>
```

这里,validator 属性指定一个方法表达式,其所指的方法称为独立的验证方法。当组件的被提交值通过转换处理、需要进一步验证时,JSF 框架将调用该独立的验证方法,对组件值进行验证处理。独立的验证方法的方法签名必须如下:

```
public void <方法名>(FacesContext context,UIComponent component,Object value)
```

方法可以抛出一个不受检查的运行时 ValidatorException 例外。

说明:事实上,JSF 框架在处理 EditableValueHolder 型组件标记的 validator 属性时,会自动创建一个类型为 javax.faces.validator.MethodExpressionActionListener 的验证器,并注册在组件上。该验证器的 validate 方法将调用 validator 属性指定的方法。

3. 直接引用验证器

让组件标记的 f:validator 子标记的 binding 属性指向某受管 bean 的一个属性,而该属性的值是一个验证器对象。此种方式不要求相关的验证器类已经注册。

下面是使用该种方式引用验证器的示例,其中 f:validator 子标记的 binding 属性通过值表达式指向 myBean 受管 bean 的 vdt 属性。

```
<h:inputText>
    <f:validator binding="#{myBean.vdt}"/>
</h:inputText>
```

myBean 受管 bean 的 vdt 属性的类型应该是一个验证器类,并能返回该种类型的一个验证器对象。

4. 使用专用标记引用标准验证器

各标准验证器都有相应的专用引用标记。使用专用引用标记,可以通过其相关属性设

置所需的验证参数，下面举例说明。

下面标记同样要求输入值不为空。

```
<h:inputText>
    <f:validateRequired/>
</h:inputText>
```

下面标记要求组件值必须是整型、且其值在 0 至 100 之间。

```
<h:inputText>
    <f:validateLongRange minimum="0" maximum="100"/>
</h:inputText>
```

下面标记要求输入的组件值(String 型)的长度应大于等于 6、小于等于 15。

```
<h:inputText>
    <f:validateLength minimum="6" maximum="15"/>
</h:inputText>
```

下面标记要求组件的输入值只能是“男”或“女”。

```
<h:inputText>
    <f:validateRegex pattern="[男女]"/>
</h:inputText>
```

上面介绍的引用验证器的方式中，方式 2 适用于自定义的独立验证方法，方式 4 是标准验证器专用的。其他两种方式既适用于标准验证器，也适用于后面要介绍的自定义验证器。

7.5.3 验证错误

在请求处理生命周期的“处理验证”阶段，当一个组件的验证处理抛出例外时，通常会产生一个错误消息(FacesMessage 对象)放入消息队列，之后其他的验证器、其他组件的转换和验证处理还会继续进行。当该阶段结束时，如果之前曾有例外抛出，那么就会跳过其他阶段，直接进入呈现响应阶段。呈现的页面中会包含那些验证出错的输入值，用户可以重新修改输入。另外，页面文件中通常应包含相应的 h:message 或 h:messages 组件标记，以便把验证过程中产生错误消息呈现出来。

JSF 框架提供一组标准验证错误消息资源，每个消息资源包含一个 ID 和一个默认的消息文本，如表 7-6 所示。标准验证器在验证处理中发生错误时，都会以其中某个消息资源文本为模板产生错误消息，消息文本中的参数会用相应的具体内容代替。

表 7-6 标准验证错误消息

消息 ID	消息文本	说明
javax.faces.component.UIInput.REQUIRED	{0}: Validation Error: Value is required	{0}为组件标签
javax.faces.validator.DoubleRangeValidator.MAXIMUM	{1}: Validation Error: Value is greater than allowable maximum of "{0}"	{1}为组件标签，{0}为最大值。仅指定了最大值时使用

续表

消息 ID	消 息 文 本	说　　明
javax.faces.validator.DoubleRangeValidator.MINIMUM	{1}: Validation Error: Value is less than allowable minimum of "{0}"	{1}为组件标签,{0}为最小值。仅指定了最小值时使用
javax.faces.validator.DoubleRangeValidator.NOT_IN_RANGE	{2}: Validation Error: Specified attribute is not between the expected values of {0} and {1}.	{2}为组件标签,{0}为最小值,{1}为最大值
javax.faces.validator.DoubleRangeValidator.TYPE	{0}: Validation Error: Value is not of the correct type	{0}为组件标签。不能转换为 Double 型时使用
javax.faces.validator.LongRangeValidator.MAXIMUM	{1}: Validation Error: Value is greater than allowable maximum of "{0}"	{1}为组件标签,{0}为最大值。仅指定了最大值时使用
javax.faces.validator.LongRangeValidator.MINIMUM	{1}: Validation Error Value is less than allowable minimum of "{0}"	{1}为组件标签,{0}为最小值。仅指定了最小值时使用
javax.faces.validator.LongRangeValidator.NOT_IN_RANGE	{2}: Validation Error: Specified attribute is not between the expected values of {0} and {1}.	{2}为组件标签,{0}为最小值,{1}为最大值
javax.faces.validator.LongRangeValidator.TYPE	{0}: Validation Error: Value is not of the correct type	{0}为组件标签。不能转换为 Long 型时使用
javax.faces.validator.LengthValidator.MAXIMUM	{1}: Validation Error: Value is greater than allowable maximum of "{0}"	{1}为组件标签,{0}为最大长度
javax.faces.validator.LengthValidator.MINIMUM	{1}: Validation Error: Value is less than allowable minimum of "{0}"	{1}为组件标签,{0}为最小长度

与标准转换错误消息不同,各标准验证错误消息的概要文本和详细文本都是相同的。

如果对组件的 requiredMessage、validatorMessage 属性进行了设置,那么这些属性的值将代替由相关验证器或独立验证方法产生的错误消息显示于响应页面。

也可以通过自定义消息包来覆盖上述标准验证错误消息文本,即让标准验证器基于开发人员指定的消息文本产生验证错误消息。该技术将在第 9 章详细介绍。

7.6　自定义验证器

标准验证器能满足一些基本的验证需求,但很多与应用相关的验证任务还需要自定义验证器来完成。本节介绍自定义验证器类的编写及注册。

7.6.1　编写自定义验证器类

验证器类必须实现验证器接口,即 javax.faces.validator.Validator 接口。Validator 接

口仅声明了一个所谓验证方法，用于检验组件值的正确性。下面是其方法签名与说明。

```
public void validate(FacesContext context, UIComponent component, Object value)
                    throws ValidatorException
```

方法包含三个参数，当 JSF 框架要调用该方法对一个组件值进行验证时，会自动传入这些参数。

- context：当前请求上下文，保存了与当前请求处理相关的所有信息。
- component：正被验证的 UI 组件。
- value：要被验证的组件值。

如果验证失败，方法可以抛出一个不受检查的运行时 ValidatorException 例外。该例外在创建时通常应包含一个 FacesMessage 消息，JSF 框架在处理例外时，会将该消息压入消息队列。当所在阶段结束时，控制将直接进入"呈现响应"阶段，相应的消息可以被显示在页面上。

7.6.2 注册自定义验证器类

引用验证器之前通常应先注册相应的验证器类。验证器类按 ID 注册。前面已经介绍，标准验证器类在 JSF 应用启动时会自动完成注册，但自定义验证器类则需要开发人员自行注册。注册验证器类的方法有两种：一是在 Faces 配置文件中用 validator 元素进行声明，validator 是 faces-config 元素的子元素；二是用@FacesValidator 型标注修饰验证器类。

下面是在 Faces 配置文件中注册验证器类的一个示例。

```
<faces-config ......>
    <validator>
        <validator-id>myvalidator</validator-id>
        <validator-class>validator.MyValidator</validator-class>
    </validator>
</faces-config>
```

其中 validator-id 子元素指定注册的验证器类的注册 ID，validator-class 子元素指定该验证器类的完整类名。

下面是采用标注方式注册验证器类的示例，此时只需在标注中指定注册 ID。

```
@FacesValidator("myvalidator")
public class MyValidator implements Validator{...}
```

需要时，可以用自定义验证器替换标准验证器。例如，若在注册 validator. MyValidator 验证器类时，指定注册 ID 为 javax. faces. Length（为标准验证器类 LengthValidator 的注册 ID），那么当按以下方式引用验证器时：

```
<f:validateLength/>
```

或

```
<f:validator validatorId="javax.faces.Length"/>
```

实际引用的将不再是标准验证器 LengthValidator，而是上述自定义验证器。

7.6.3 自定义验证器应用示例

本例继续第 7.3.3 小节中介绍的 JSF 应用，为其添加一个自定义验证器，用于检验用户输入的 ISBN 编号是否正确。ISBN 编号通常由 13 位数字组成，其中第 13 位（即最后一位）是校验码。校验码的计算方法如下：

（1）分别用 1 乘以编号前面 12 位中的各奇数位、3 乘以各偶数位；

（2）将上面所有的乘积相加得到总和，然后再用该总和除以 10 得到一个余数；

（3）用 10 减去上面得到的余数得到一个差，如果差为 10，则校验码为 0，否则差即为校验码。

下面介绍具体的实现方法。首先创建 Java 包 validator，并在包中创建自定义验证器类 ISBNValidator.java，具体代码如代码清单 7-6 所示。

代码清单 7-6　自定义验证器类（ISBNValidator.java）

```
package validator;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class ISBNValidator implements Validator {
  public void validate(FacesContext context,UIComponent component,Object value)
                                   throws ValidatorException {
    StringBuilder sb=new StringBuilder(String.valueOf(value));
    boolean success=true;
    if(sb.length()!=13){
      success=false;
    } else {
      int sum=0;
      for(int i=0;i<11;i=i+2){
        sum=sum +1 * (sb.charAt(i)-48) +3 * (sb.charAt(i+1)-48);
      }
      System.out.println("sum="+sum);
      int c=10 -sum%10;
      if(c==10) c=0;
      if(sb.charAt(12)-48!=c) success=false;
    }
    if(!success){
      String s="ISBN 编号错误,请重新输入!";
      FacesMessage msg=new FacesMessage(s,s);
      throw new ValidatorException(msg);
    }
  }
}
```

然后注册该验证器类，在 Faces 配置文件(faces-config. xml)中进行声明，见代码清单 7-7。

代码清单 7-7　注册自定义验证器类(validator. ISBNValidator)

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.0" ……>
  ……
  <validator>
    <validator-id>isbnvalidator</validator-id>
    <validator-class>validator.ISBNValidator</validator-class>
  </validator>
</faces-config>
```

最后在欢迎页面(index. xhtml)的文本域组件标记中用 f:validator 子标记引用已注册的自定义验证器，见代码清单 7-8。

代码清单 7-8　在页面 index. xhtml 中引用自定义验证器

```
<h:inputText id="isbn" value="#{myBean.isbn}">
  ……
  <f:validator validatorId="isbnvalidator"/>
</h:inputText>
```

7.7　小　　结

- 转换器模型的编程主要涉及定义转换器类、注册转换器类和引用转换器等三方面的工作。
- 标准转换器由 JSF 规范定义、随 JSF 实现一起提供给用户，主要用于一些常见的 Java 类型的组件值与其字符串文本表示之间的转换。
- 转换器类可按 ID 注册，也可按类型注册。所有的标准转换器都自动完成按 ID 注册或者按类型注册。
- 引用转换器的方式包括：默认方式(基于类型)、通过 ID 引用、直接引用以及使用专用标记引用特定的标准转换器。
- 自定义转换器类必须实现 Converter 接口，提供 getAsObject 方法和 getAsString 方法的实现。
- 验证器模型的编程主要涉及定义验证器类、注册验证器类和引用验证器等三方面的工作。
- 标准验证器由 JSF 规范定义、随 JSF 实现一起提供给用户，主要用于完成一些基本的组件值验证处理功能。
- 验证器类只按 ID 进行注册。所有标准验证器类都自动完成注册。
- 引用验证器的方式包括：通过 ID 引用、引用独立的验证方法、直接引用以及使用专用标记引用特定的标准验证器。
- 自定义验证器类必须实现 Validator 接口，提供 validate 方法的实现。

习　题　7

1．转换器和验证器的作用是什么？

2．结合转换器和验证器，简述“处理验证”阶段的基本过程。

3．注册转换器类和验证器类的方式有哪几种？标准转换器类和标准验证器类的注册情况如何？

4．简述引用转换器和验证器的各种方式及其特点。

5．如何自定义一个转换器？如何自定义一个验证器？

6．修改应用项目 sh4_logandreg（第 4 章习题 5）中的登录功能：编写一个自定义验证器，用以验证用户输入的用户名和密码是否合法。同时对原来的动作方法做必要的修改，即删去有关上述验证功能的代码。

第 8 章　JSF 事件处理

本章主题：

- JSF 事件处理概述
- 动作事件及其处理
- 值变化事件及其处理
- 阶段事件及其处理
- 系统事件及其处理

JSF 是一种以组件为中心、事件驱动编程模型为基础的 Web 应用的用户界面软件框架。了解 JSF 的事件处理机制，掌握事件处理编程技术，对 JSF 应用开发人员来说是必不可少的。

8.1　JSF 事件处理概述

JSF 事件模型基于 JavaBean 事件模型，简称事件—监听器模型。事件泛指对象的某种状态变化，其中的对象称为事件源。当一个事件引发时，事件源能够将该事件通告给对这种事件感兴趣的监听器。监听器接收到通告后，可以了解事件的属性，并可根据需要进行相应的处理和响应，如图 8-1 所示。一个监听器若对某个事件源的某种事件感兴趣，应该先进行注册。

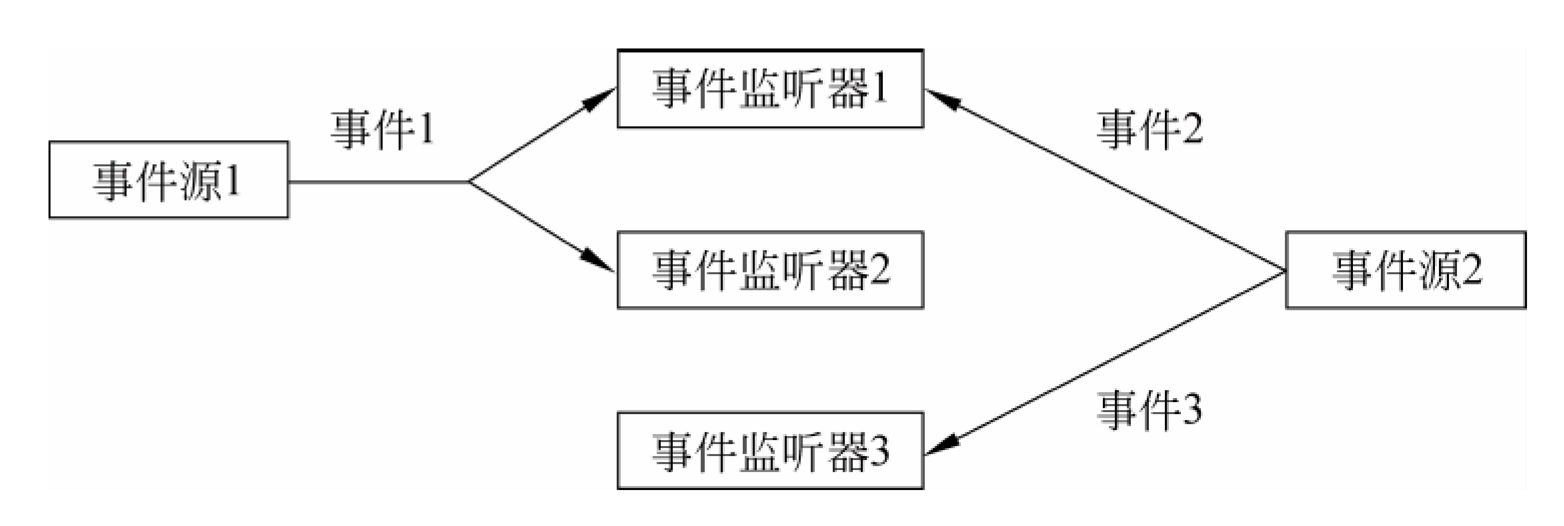


图 8-1　事件—监听器模型

在事件—监听器模型中，一个事件源可以引发多种类型的事件，如图中事件源 2 可以引发两种类型的事件(事件 2 表示某种类型的事件，事件 3 表示另一种类型的事件)。对一个事件源引发的某种类型的事件，可以有多个事件监听器感兴趣，如图中事件监听器 1 和事件监听器 2 都对事件源 1 的一种事件类型感兴趣，当这种类型的一个事件引发时，事件源 1 将依次通告给上述两个监听器。一个监听器也可以对多个事件源引发的事件感兴趣，如图中事件监听器 1 既对事件源 1 引发的事件、也对事件源 2 引发的事件感兴趣。

1. 事件与事件源

事件用事件类的实例表示。在 JSF 中，事件分三大类：Faces 事件、阶段事件和系统事件，如图 8-2 所示。图中除 EventObject 属于 java. util 包，其他事件类都属于 javax. faces.

event 包。

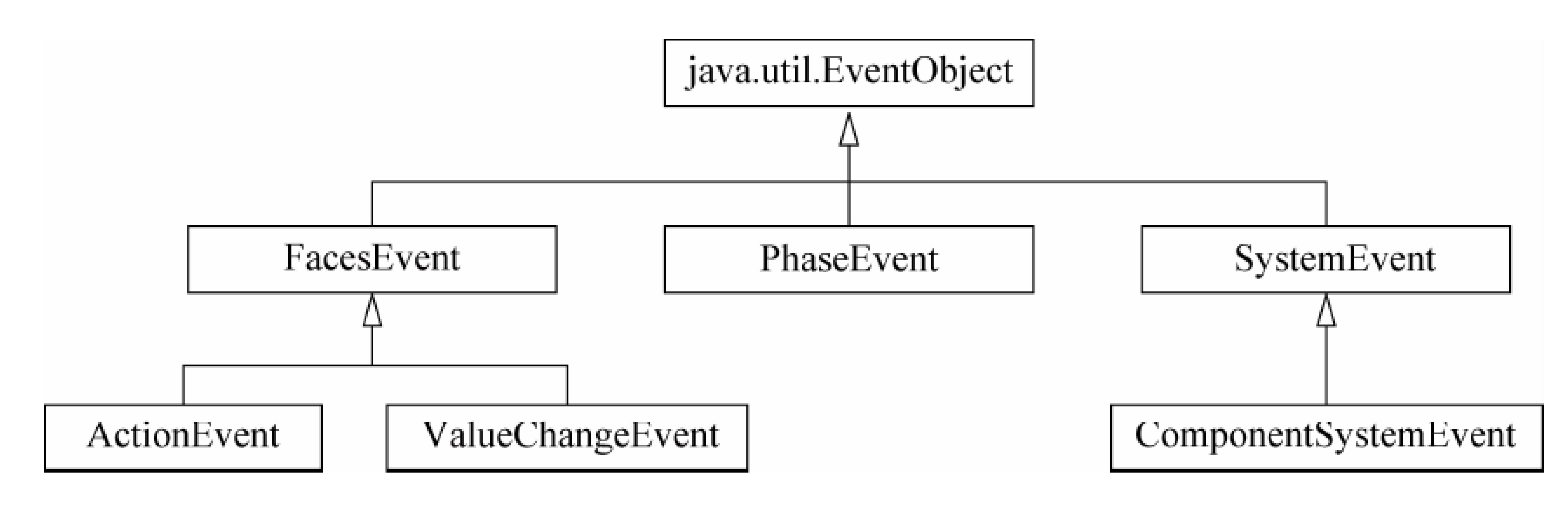


图 8-2　JSF 事件类型

Faces 事件又称应用事件，由用户界面组件引发和广播。Faces 事件的基类是 FacesEvent，包括动作事件（ActionEvent）和值变化事件（ValueChangeEvent）。动作事件由 UICommand 组件引发，包括 HtmlCommandButton 和 HtmlCommandLink 组件。动作事件表示用户在界面上单击这些组件呈现的递交按钮或超链接。值变化事件由 UIInput 组件引发，包括基本输入类组件、选择类组件和视图参数组件。值变化事件表示用户为某个 UIInput 组件指定了一个新值。

阶段事件的事件源是 Lifecycle 型对象。阶段事件表示 JSF 请求处理生命周期中某个阶段的开始或结束。

与阶段事件类似，系统事件表示 JSF 应用在运行期间的某个特定的时间点，如 JSF 应用启动完成、某组件已被添加到视图等。系统事件是一类事件的总称。具体的系统事件类型应该扩展 SystemEvent 抽象类，其事件源可以是某种任意类型对象。

组件系统事件是事件源为某种 UIComponent 对象的系统事件。具体的组件系统事件类型应该扩展 ComponentSystemEvent 抽象类。

2. 监听器与监听器接口

监听器是监听器类的实例，包含用于处理事件的方法。所谓事件源将事件通告给监听器，指的就是调用监听器中的事件处理方法。

一般来说，一个监听器接口对应于某种事件类型，声明了用于处理该种事件的方法签名。一个具体的监听器类必须实现监听器接口，提供接口中声明的方法的实现。图 8-3 给出了 JSF 规范包含的一些主要监听器接口及其继承关系。

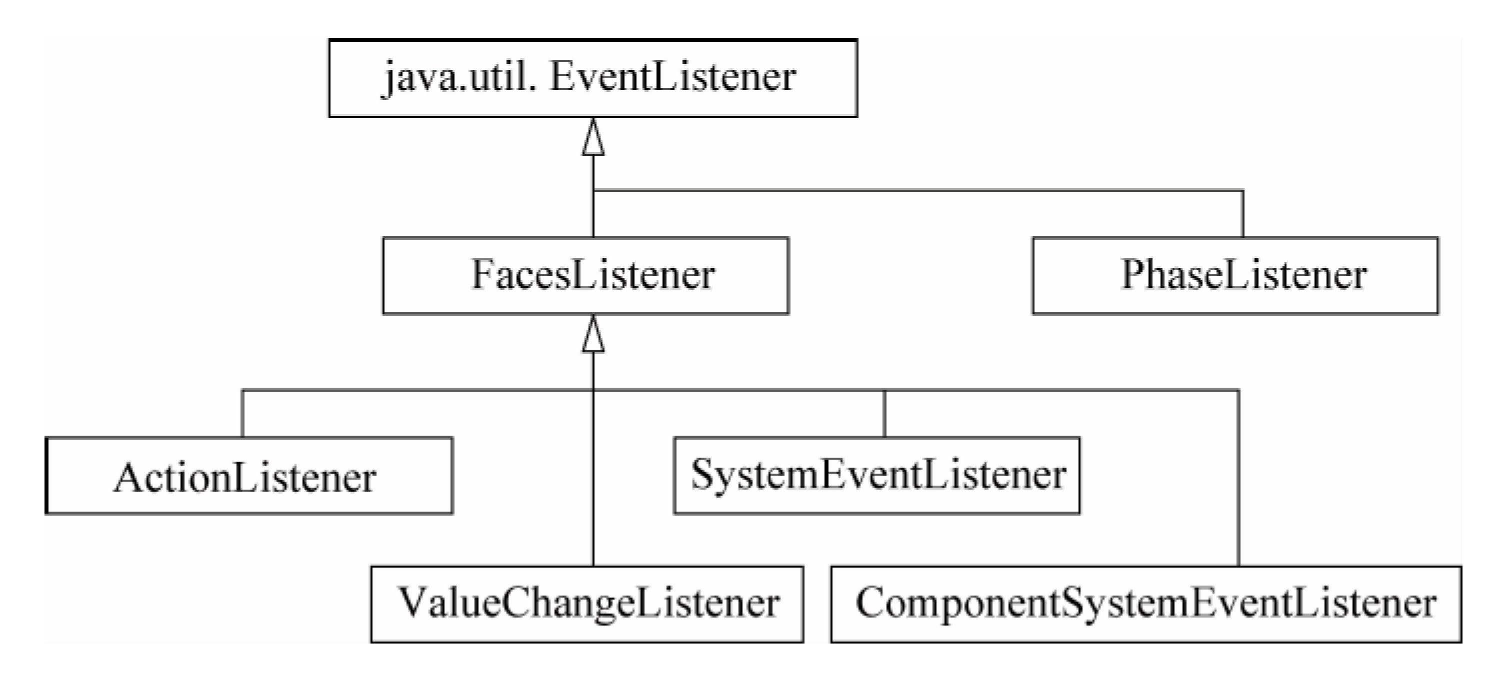


图 8-3　JSF 监听器接口

这里，ActionListener 型监听器用于监听、处理动作事件；ValueChangeListener 型监听

器用于监听、处理值变化事件;PhaseListener 型监听器用于监听、处理阶段事件。

如上所述,通常一种事件类型对应一个监听器接口,但对系统事件和组件系统事件并非如此。系统事件和组件系统事件都是一类事件的总称。但不管是哪种具体的系统事件类型,其监听器类都应该实现 SystemEventListener 接口。对于具体的组件系统事件类型,其相应监听器类则可以通过实现 ComponentSystemEventListener 接口来定义。

要掌握事件处理编程,首先要了解各种事件类型及其事件源,其次要能够扩展监听器接口定义监听器类和监听器方法,最后要知道如何向事件源注册监听器。

8.2 动作事件及其处理

本节介绍动作事件的含义、引发机制,以及动作事件的监听与处理。

8.2.1 动作事件

动作事件用 ActionEvent 事件类的实例对象表示。动作事件由实现了 ActionSource 接口(属于 javax.faces.component 包)的 UICommand 组件(也称为动作组件)引发。在 JSF 规范中,这类组件包括 HtmlCommandButton 和 HtmlCommandLink 组件。

上述两个组件在用户界面中分别呈现为动作按钮或动作超链接。当用户单击动作按钮或动作超链接时,将产生一个 POST 回送请求,其中的请求参数就包含该动作按钮或动作超链接的值本身。该请求将由 JSF 框架处理。在请求处理生命周期的“应用请求值”阶段,若 JSF 框架检测到请求参数有某动作按钮或动作超链接的值,表明该动作按钮或动作超链接已被用户激活,此时 JSF 框架将为相应的 HtmlCommandButton 或 HtmlCommandLink 组件创建一个动作事件。

动作事件的创建需用到 ActionEvent 类中的以下构造方法,其中传入的组件参数应该是实现 ActionSource 接口的 HtmlCommandButton 组件或 HtmlCommandLink 组件。

```
public ActionEvent(UIComponent component)
```

动作事件具有以下实例方法,可以返回该事件的事件源,也即在创建动作事件时指定的动作按钮组件或动作超链接组件。

```
public UIComponent getComponent()
```

动作事件创建后并不会马上处理,而是被先放到事件队列中。直到整个“应用请求值”阶段结束或进入“调用应用”阶段,动作事件才会从事件队列中取出并被处理。

默认情况下(组件的 immediate 属性值为 false),动作事件将在“调用应用”阶段被广播至已在动作组件上注册对动作事件感兴趣的监听器。如果组件的 immediate 属性值为 true,那么动作事件会在“应用请求值”阶段结束时被广播至已注册的对动作事件感兴趣的监听器。

8.2.2 动作监听器

对动作事件感兴趣的监听器称为动作监听器,是实现 ActionListener 接口的监听器类的实例。ActionListener 接口仅声明了一个用于处理动作事件的 processAction 方法。

```
public void processAction(ActionEvent event) throws AbortProcessingException
```

AbortProcessingException 是定义于 javax.faces.event 包的一个不受检查的运行时例外。方法抛出该例外的作用是通知 JSF 框架：该事件不必做进一步的处理，即不需要再广播至其他的监听器。

下面代码是一个简单的动作监听器类的示例，其中用于处理动作事件的 processAction 方法象征性地输出了一条信息。

```
package listener;
import javax.faces.event.ActionListener;
import javax.faces.event.ActionEvent;
public class MyActionListener implements ActionListener {
    public void processAction(ActionEvent ae){
        System.out.println("MyActionListener is Processing an action event...");
    }
}
```

8.2.3 注册动作监听器

一个监听器要监听和处理某个组件引发的事件，必须预先在该组件上注册。每种组件都会提供一组合适的方法用以注册相应的监听器。比如 UICommand 组件的下面方法就用以在组件上注册一个动作监听器。

```
public void addActionListener(ActionListener listener)
```

在 JSF 中，更常用的方法是采用声明的方式在页面文件中注册所需的动作监听器。具体做法是：在引发动作事件的动作组件标记中嵌入 f:actionListener 核心标记。

```
<h:commandButton value="OK" action="#{myBean.m}">
    <f:actionListener type="listener.MyActionListener"/>
</h:commandButton>
```

上面代码中，f:actionListener 标记 type 属性指定监听器类的完整类名。当 JSF 框架处理视图时会在其父标记组件(即动作按钮组件)上注册一个指定类型的监听器对象。

可以在动作按钮组件标记或动作超链接组件标记中嵌入多个 f:actionListener 标记，达到注册多个监听器的目的。如果注册了多个监听器，引发的动作事件将按注册的顺序(即声明的顺序)依次通告给各动作监听器。

也可以通过动作组件标记的 actionListener 属性注册一个独立的动作监听方法。下面代码演示了这种方法的使用。

```
<h:commandButton value="OK" actionListener="#{myBean.method}" action=
    "#{myBean.m}"/>
```

这里，actionListener 属性指定一个方法表达式，其所指的方法称为独立的动作监听方法。当动作事件引发时，该动作监听方法将被通告(调用)。动作监听方法的方法签名应该如下：

```
public void <方法名> (ActionEvent ae)
```

或

```
public void <方法名> ()
```

说明：事实上，JSF 框架在处理动作组件标记的 actionListener 属性时，会自动创建一个类型为 javax.faces.event.MethodExpressionActionListener 的监听器，并注册在动作组件上。该监听器的 processAction 方法将调用 actionListener 属性指定的方法。

对一个动作组件，除了通过 f:actionListener 子标记和 actionListener 属性注册的监听器和独立监听方法外，JSF 框架还会提供一个默认的监听器，该默认监听器的 processAction 方法会调用 action 属性指定的动作方法，并以动作方法返回的结果值为参数、调用导航处理器的导航方法进行导航处理。

对一个动作组件，如果既注册了动作监听器（可以有多个），又指定了动作监听方法和动作方法，那么将按以下顺序通告和调用：

- actionListener 属性指定的动作监听方法。
- f:actionListener 标记指定的监听器（按声明顺序）。
- action 属性指定的动作方法。

8.3　值变化事件及其处理

本节介绍值变化事件的含义、引发机制，以及值变化事件的监听与处理。

8.3.1　值变化事件

值变化事件用 ValueChangeEvent 事件类的实例对象表示。值变化事件的事件源是实现了 EditableValueHolder 接口（属于 javax.faces.component 包）的 UIInput 组件。在 JSF 规范中，这类组件包括基本输入类组件、选择类组件和视图参数组件等。

如果一个请求包含与某个 UIInput 组件对应的请求参数，那么该请求参数将被该组件读入，然后进行类型转换和验证处理。当一个 UIInput 组件的值通过了验证、且其新值不同于原有的值，JSF 框架将为相应的 UIInput 组件创建一个值变化事件。

值变化事件的创建需用到 ValueChangeEvent 类中的以下构造方法，其中传入的第一个组件参数应该是引发该事件的具体的 UIInput 组件；第二个参数应该是组件原有的值；第三个参数应该是组件的新值。

```
public ValueChangeEvent(UIComponent component, Object oldValue, Object newValue)
```

值变化事件具有以下实例方法，可以返回该事件的事件源，以及该事件源组件的原值和新值。

- public UIComponent getComponent()。
- public Object getOldValue()。
- public Object getNewValue()。

值变化事件创建后并不会马上处理，而是被先放到事件队列中。直到验证处理所在阶段结束，值变化事件才会从事件队列中取出并被处理。

默认情况下(组件的 immediate 属性值为 false)，值变化事件将在“处理验证”阶段结束时被广播至已在组件上注册对值变化事件感兴趣的监听器。如果组件的 immediate 属性值为 true，那么组件值的验证会在“应用请求值”阶段进行，相应地，该组件引发的值变化事件将在“应用请求值”阶段结束时被广播至已注册的对值变化事件感兴趣的监听器。

8.3.2 值变化监听器

对值变化事件感兴趣的监听器称为值变化监听器，是实现 ValueChangeListener 接口的监听器类的实例。ValueChangeListener 接口仅声明了一个用于处理值变化事件的方法。

```
public void processValueChange(ValueChangeEvent event) throws AbortProcessingException
```

AbortProcessingException 是一个不受检查的运行时例外。方法抛出该例外的作用是通知 JSF 框架：该事件不必做进一步的处理，即不需要再广播至其他的监听器。

下面代码是一个简单的值变化监听器类的示例，其中 processValueChange 方法只是象征性地输出了事件源组件的值的变化。

```
package listener;
import javax.faces.event.ValueChangeEvent;
import javax.faces.event.ValueChangeListener;
public class MyValueChangeListener implements ValueChangeListener {
    public void processValueChange(ValueChangeEvent event){
        System.out.println("Event Source:"+event.getComponent());
        System.out.println("Old Value:" +event.getOldValue());
        System.out.println("New Value:" +event.getNewValue());
    }
}
```

8.3.3 注册值变化监听器

值变化监听器要监听和处理某个 UIInput 组件引发的值变化事件，必须预先在该组件上注册。UIInput 组件提供以下方法用以注册一个值变化监听器。

```
public void addValueChangeListener(ValueChangeListener listener)
```

在 JSF 中，采用声明的方式注册值变化监听器是一种更常用的方法。具体做法是：在引发值变化事件的 UIInput 组件标记中嵌入 f:valueChangeListener 核心标记。

```
<h:inputText value="#{myBean.degree}">
    <f:valueChangeListener type="listener.MyValueChangeListener"/>
</h:inputText>
```

这里，f:valueChangeListener 标记的 type 属性指定值变化监听器类的完整类名。当 JSF 框架处理该组件标记时会在其父 UIInput 组件(上面示例中即为 HtmlInputText 组件)上注

册一个指定类型的监听器对象。

可以在某种 UIInput 组件标记中嵌入多个 f:valueChangeListener 标记,达到注册多个值变化监听器的目的。如果注册了多个监听器,引发的值变化事件将按注册的顺序(即声明的顺序)依次通告给各值变化监听器。

也可以通过某种 UIInput 组件标记的 valueChangeListener 属性注册一个独立的值变化监听方法。下面是使用该属性的一个示例。

```
<h:inputText value="#{myBean.degree}" valueChangeListener="#{myBean.method}"/>
```

这里,valueChangeListener 属性指定一个方法表达式,其所指的方法称为值变化监听方法。当值变化事件引发时,该值变化监听方法将被通告(调用)。值变化监听方法的方法签名应该如下:

```
public void <方法名>(ValueChangeEvent vce)
```

或

```
public void <方法名>()
```

说明:事实上,JSF 框架在处理某种 UIInput 组件标记的 valueChangeListener 属性时,会自动创建一个类型为 javax.faces.event.MethodExpressionValueChangeListener 的监听器对象,并注册在该 UIInput 组件上。该监听器的 processValueChange 方法将调用 valueChangeListener 属性指定的方法。

对一个 UIInput 组件,如果既注册了值变化监听器(可以有多个),又指定了值变化监听方法,那么将按以下顺序通告和调用:

- valueChangeListener 属性指定的值变化监听方法。
- f:valueChangeListener 标记指定的监听器(按声明顺序)。

8.3.4 值变化事件应用示例

该应用项目(ch8_valuechange)包含一个 JSF 页面、一个受管 bean 和三个 Java 类。应用的运行效果如图 8-4 所示。



图 8-4 应用 ch8_valuechange 运行效果图

当用户从"部门名称"单选菜单中选择不同部门(值变化)时,"教师姓名"单选菜单中的选项会发生相应的变化,即变成了指定部门的所有教师的姓名。当用户单击"确认"按钮时,

页面下方会显示所选教师的职工号。

1. 模型

该应用的模型包括三个 Java 类，都属于 model 包。第一个类是表示部门的 Department 类，见代码清单 8-1。其中包含两个可读的实例变量，实例变量相应的 getter 方法在代码清单中被省略了。

代码清单 8-1 model. Department 类

```
package model;

public class Department {
  private String did;                     //部门编号
  private String dname;                   //部门名称

  public Department(){
  }
  public Department(String did,String dname){
    this.did=did;
    this.dname=dname;
  }
  ...                                     //实例变量相应的 getter 方法
}
```

第二个类是表示教师的 Teacher 类，见代码清单 8-2。其中包含三个可读的实例变量，实例变量相应的 getter 方法在代码清单中被省略了。

代码清单 8-2 model. Teacher 类

```
package model;

public class Teacher {
  private String tid;                     //职工号
  private String tname;                   //教师姓名
  private String did;                     //所属部门

  public Teacher(){
  }
  public Teacher(String tid,String tname,String did){
    this.tid=tid;
    this.tname=tname;
    this.did=did;
  }
  ...                                     //实例变量相应的 getter 方法
}
```

最后一个类是模拟业务数据的 DataBase 类，见代码清单 8-3。该类定义了两个类变量，其中 depts 引用一个部门表，teachers 引用一个教师表，这两个表在类装入时被初始化。

代码清单 8-3　model.DataBase 类

```java
package model;
import java.util.ArrayList;
import java.util.List;

public class DataBase {
  private final static List<Department>depts=new ArrayList<Department>();
  private final static List<Teacher>teachers=new ArrayList<Teacher>();
  static {
    depts.add(new Department("01","外语系"));
    depts.add(new Department("02","信息学院"));
    teachers.add(new Teacher("0001","李小明","02"));
    teachers.add(new Teacher("0002","吴英","01"));
    teachers.add(new Teacher("0003","刘京菁","02"));
    teachers.add(new Teacher("0004","赵永义","01"));
    teachers.add(new Teacher("0005","郝杰","02"));
  }
  //返回所有的部门
  public static List<Department>getDepartments(){
    return depts;
  }
  //根据部门编号 did 返回该部门的所有教师
  public static List<Teacher>getTeachers(String did){
    List<Teacher>list=new ArrayList<Teacher>();
    for(Teacher t:teachers){
      if(t.getDid().equals(did)) list.add(t);
    }
    return list;
  }
}
```

2. 受管 bean

该应用定义了一个视图作用域的受管 bean 类，即 bean.Index，见代码清单 8-4。其中包含两个只读属性和两个可读写属性，这些属性相应的 getter 方法和 setter 方法在代码清单中被省略了。

代码清单 8-4　受管 bean（bean.Index）

```java
package bean;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;
import javax.faces.context.FacesContext;
```

```
import javax.faces.event.ValueChangeEvent;
import javax.faces.model.SelectItem;
import model.DataBase;
import model.Department;
import model.Teacher;

@ManagedBean
@ViewScoped
public class Index implements Serializable{
  //只读属性,为“部门名称”单选菜单提供选项
  private List<SelectItem>depts=new ArrayList<SelectItem>();
  //只读属性,为“教师姓名”单选菜单提供选项
  private List<SelectItem>teachers;
  //可读写属性,存放“部门名称”单选菜单的值,即所选部门的部门号
  private String did;
  //可读写属性,存放“教师姓名”单选菜单的值,即所选教师的职工号
  private String tid;

  public Index(){
    SelectItem item=new SelectItem();
    item.setLabel("请选择...");
    item.setNoSelectionOption(true);
    depts.add(item);
    List<Department>ds=DataBase.getDepartments();
    for(Department d : ds){
      depts.add(new SelectItem(d.getDid(),d.getDname()));
    }
  }
  public void valueChange(ValueChangeEvent vce){
    String did=(String)vce.getNewValue();
    teachers=new ArrayList<SelectItem>();
    SelectItem item=new SelectItem();
    item.setLabel("请选择...");
    item.setNoSelectionOption(true);
    teachers.add(item);
    tid=null;
    if(did==null) return;
    List<Teacher>ts=DataBase.getTeachers(did);
    for(Teacher t : ts){
      teachers.add(new SelectItem(t.getTid(),t.getTname()));
    }
    FacesContext.getCurrentInstance().renderResponse();
  }
  ......              //只读属性相应的 getter 方法
```

```
...              //可读写属性相应的getter方法和setter方法
}
```

bean 实例创建时,depts 属性被初始化,且在整个视图作用域内是不变的。teachers 属性值在 valueChange 方法中被创建和初始化。valueChange 方法是一个值变化监听方法,每当用户选择不同的部门时,该方法会被调用。该方法最后一行代码(renderResponse 方法)的功能是:一旦当前阶段完成,就绕过 JSF 请求处理生命周期的的其他阶段,直接转入“呈现响应”阶段。

3. JSF 页面

该应用仅包含一个 JSF 页面,即 index.xhtml,见代码清单 8-5。下面,主要讨论“部门名称”单选菜单标记。

首先,该标记的 onchange 属性指定为 submit(),它使得当用户选择不同部门时会引起表单的提交和请求。这种提交和请求与单击“确认”按钮引起的表单提交和请求是类似的,都会经历 JSF 请求处理生命周期。但不同的是,前者产生的请求没有包含“确认”动作标记相应的请求参数,所以不会产生动作事件和执行动作方法。

其次,该标记的 valueChangeListener 属性指定受管 bean 的 valueChange 方法为其值变化事件的监听方法。无论何种方式引起表单提交和请求,如果组件值通过转换和验证,且“部门名称”值发生变化,受管 bean 的 valueChange 方法就会被调用。

最后,该标记的 immediate 属性设置为 true,这使得“部门名称”值的转换、验证和值变化事件的监听处理都将在“应用请求值”阶段进行,而不会像其他输入类组件那样在“处理验证”阶段进行。

综合以上各点,再考虑到 valueChange 方法的最后一行代码是绕过 JSF 请求处理生命周期的其他阶段、直接转入呈现响应阶段,可以得出,当用户选择不同部门而引起表单提交和请求时,服务器端所做的主要工作就是在“应用请求值”阶段调用 valueChange 方法、重新设置“教师姓名”单选菜单的选项(即 teachers 属性值),而不会进入“处理验证”、“更新模型值”等阶段,不会对其他组件值进行转换和验证处理。

代码清单 8-5　index.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>值变化事件示例</title>
  </h:head>
  <h:body>
    <h:form>
      <h:outputLabel for="s1" value="部门名称:"/>
      <h:selectOneMenu id="s1" value="#{index.did}" hideNoSelectionOption="true"
                valueChangeListener="#{index.valueChange}"
```

```
                immediate="true"
                onchange="submit()">
        <f:selectItems value="#{index.depts}"/>
      </h:selectOneMenu>
      <h:outputLabel for="s2" value="教师姓名:" style="margin-left: 10px"/>
      <h:selectOneMenu id="s2" value="#{index.tid}" hideNoSelectionOption="true">
        <f:selectItems value="#{index.teachers}"/>
      </h:selectOneMenu>
      <h:commandButton value="确认" style="margin-left: 10px"/>
    </h:form>
    <p><h:outputText value="所选教师职工号:#{index.tid}"/></p>
  </h:body>
</html>
```

8.4 阶段事件及其处理

本节介绍阶段事件的含义,以及阶段监听器类的定义与阶段监听器的注册。

8.4.1 阶段事件

阶段事件用 PhaseEvent 事件类的实例对象表示。阶段事件的事件源是 Lifecycle 型对象,该生命周期对象管理 JSF 请求处理生命周期的整个过程,负责执行生命周期的各个阶段。在请求处理生命周期中,阶段状态的变化将引发阶段事件。

阶段事件的创建需用到 PhaseEvent 类中的以下构造方法:

```
public PhaseEvent(FacesContext context, PhaseId phaseId, Lifecycle lifecycle)
```

第一个参数为与当前请求相关联的 FacesContext 对象;第二个参数是与该阶段事件相关联的请求处理阶段的阶段标识符;第三个参数是作为事件事件源的生命周期对象。

阶段事件对象具有以下实例方法,可以返回该事件的事件源、阶段标识符以及请求处理上下文对象。

- Object getSource()。
- PhaseId getPhaseId()。
- FacesContext getFacesContext()。

与动作事件和值变化事件不同,阶段事件引发后会被立刻广播给已注册的、对阶段事件感兴趣的监听器。

8.4.2 阶段监听器

对阶段事件感兴趣的监听器称为阶段监听器,是实现 PhaseListener 接口的监听器类的实例。PhaseListener 接口声明了以下方法。

(1) PhaseId getPhaseId()

返回请求处理阶段的阶段标识符,当前监听器仅对该阶段的阶段事件感兴趣。

PhaseId 类定义了此 getPhaseId 方法可以返回的各有效值:

- PhaseId.ANY_PHASE。
- PhaseId.RESTORE_VIEW。
- PhaseId.APPLY_REQUEST_VALUES。
- PhaseId.PROCESS_VALIDATIONS。
- PhaseId.UPDATE_MODEL_VALUES。
- PhaseId.INVOKE_APPLICATION。
- PhaseId.RENDER_RESPONSE。

如果方法返回 PhaseId.ANY_PHASE,表示当前监听器对所有阶段的阶段事件都感兴趣。

(2) void beforePhase(PhaseEvent event)

在监听器感兴趣的阶段的开始处,该方法被通告(调用)。

(3) void afterPhase(PhaseEvent event)

在监听器感兴趣的阶段的结尾处,该方法被通告(调用)。

下面代码是一个简单的阶段监听器类的示例。这种监听器对所有阶段的阶段事件都感兴趣,事件处理方法只是简单输出当前阶段的标识符。

```
package listener;
import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;
public class MyPhaseListener implements PhaseListener{
    @Override
    public void beforePhase(PhaseEvent pe){
        System.out.println("阶段:" +pe.getPhaseId() +"开始");
    }
    @Override
    public void afterPhase(PhaseEvent pe){
        System.out.println("阶段:" +pe.getPhaseId() +"结束");
    }
    @Override
    public PhaseId getPhaseId(){
        return PhaseId.ANY_PHASE;
    }
}
```

8.4.3 注册阶段监听器

阶段事件的事件源是 Lifecycle 型对象。Lifecycle 型对象提供以下方法用以注册一个阶段监听器。

```
public void addPhaseListener(PhaseListener listener)
```

采用声明的方式注册阶段监听器是一种更常用的方法。一般的做法是在 Faces 配置文件中,用嵌套于 lifecycle 元素的 phase-listener 子元素指定要注册的监听器。

```
<faces-config ......>
    <lifecycle>
        <phase-listener>listener.MyPhaseListener</phase-listener>
    </lifecycle>
</faces-confhg>
```

phase-listener 元素指定一个阶段监听器类的完整类名。在部署 JSF 应用时，指定类型的一个监听器实例被自动创建，并注册在生命周期对象上。

上面代码仅指定了一个监听器，可以在 lifecycle 元素嵌入多个 phase-listener 元素指定、注册多个阶段监听器。各监听器按声明的顺序注册。

当一个阶段事件引发时，对该事件感兴趣的阶段监听器将被依次通告。具体来说，各监听器的 beforePhase 方法将按各监听器的注册顺序依次被调用；各监听器的 afterPhase 方法将按各监听器注册顺序的倒序依次被调用。

对阶段监听器，还可以在视图页面中用 f:phaseListener 核心标记将其注册在视图根上。该标记可以出现在页面中根标记(即 html 标记)内的任何位置。

```
<f:phaseListener type="listener.MyPhaseListener"/>
```

与 Faces 配置文件中的 phase-listener 元素相比较，该标记只是注册局部阶段监听器，即指定的监听器只监听当前视图的请求处理过程中引发的阶段事件。而 phase-listener 元素指定的是全局监听器，这些监听器会监听当前应用中所有请求处理过程中引发的阶段事件。

如果既注册了局部监听器、又包含全局监听器，那么它们的顺序是全局监听器在前、局部监听器在后。

8.5　系统事件及其处理

本节首先介绍系统事件的概念、特点以及一些具体的系统事件类型，然后介绍系统事件监听器类的定义和系统事件监听器的注册。

8.5.1　系统事件

与阶段事件类似，系统事件表示 JSF 应用在运行期间的某个特定的时间点。但与阶段事件相比，系统事件表示的时间点的范围更广、颗粒度更细，如 JSF 应用启动完成、某组件已被添加到视图等。

一个具体的系统事件是扩展 SystemEvent 抽象类的某种具体类的实例对象，其事件源可以是某种任意类型对象。一个具体的组件系统事件是扩展 ComponentSystemEvent 抽象类的某种具体类的实例对象，其事件源为某种 UIComponent 型对象。

ComponentSystemEvent 抽象类是 SystemEvent 抽象类的子类，所以组件系统事件也是一种系统事件。

表 8-1 列出了 JSF 规范提供的一些具体的系统事件类，其中前两个是普通的系统事件类，后面五个是组件系统事件类。这些事件类都属于 javax. faces. event 包。

表 8-1　系统事件与组件系统事件

事件类名	说明	事件源类型
PostConstructApplicationEvent	应用程序正好启动完毕	Application
PreDestroyApplicationEvent	应用程序即将关闭	Application
PostAddToViewEvent	组件正好被添加至视图	UIComponent
PreValidateEvent	组件即将被验证	UIComponent
PostValidateEvent	组件正好验证完毕	UIComponent
PreRenderViewEvent	视图(视图根)即将被呈现	UIViewRoot
PreRenderComponentEvent	组件即将被呈现	UIComponent

8.5.2　系统事件监听器

对系统事件感兴趣的监听器称为系统事件监听器，对组件系统事件感兴趣的监听器则可称为组件系统事件监听器。前面已经介绍，系统事件和组件系统事件都是一类事件的总称，包含一组具体的事件类型，如 PostConstructApplicationEvent 等。但不管是哪种具体的系统事件类型，其监听器类都应该实现 SystemEventListener 接口。对于具体的组件系统事件类型，其相应监听器类则可以通过实现 ComponentSystemEventListener 接口来定义。

系统事件监听器接口，即 SystemEventListener，声明了两个方法，其中 processEvent 方法在事件引发时被调用，用于处理系统事件；isListenerForSource 方法用于检测该监听器是否对指定的事件源感兴趣。

```
public interface SystemEventListener extends FacesListener {
    void processEvent(SystemEvent event) throws AbortProcessingException;
    boolean isListenerForSource(Object source);
}
```

下面是一个具体的系统事件监听器类的示例。该监听器可以监听由 Application 对象引发的各种系统事件。processEvent 方法只是示意性地输出了事件与事件源的类型名。

```
package listener;
import javax.faces.application.Application;
import javax.faces.event.SystemEvent;
import javax.faces.event.SystemEventListener;
public class MySystemEventListener implements SystemEventListener {
    public void processEvent(SystemEvent event){
        System.out.println("事件类型:"+event.getClass());
        System.out.println("事件源类型:"+event.getSource().getClass());
    }
    public boolean isListenerForSource(Object source){
        return source instanceof Application;
    }
}
```

组件系统事件监听器接口，即 ComponentSystemEventListener，仅声明了一个用于处理事件的 processEvent 方法。

```
public interface ComponentSystemEventListenerextends FacesListener {
    void processEvent(ComponentSystemEvent event) throws AbortProcessingException;
}
```

由于组件系统监听器总是注册在某个 UIComponent 上、监听由该组件引发的系统事件，所以组件系统事件监听器并不需要指定它对哪种事件源引发的事件感兴趣。

需要注意，ComponentSystemEvent 是 SystemEvent 的子类，所以一个组件系统事件可以被看作是一种普通的系统事件。此时，组件系统事件可以用普通的系统事件监听器来监听和处理。而 ComponentSystemEventListener 与 SystemEventListener 之间不存在子类与超类的关系，所以一个组件系统事件监听器不能被看作是一个普通的系统事件监听器，不能用于监听和处理普通的系统事件。

8.5.3 注册系统事件监听器

系统事件的事件源可以是某种任意类型对象，Application 对象的 subscribeToEvent 方法用以注册一个系统事件监听器。

```
public void subscribeToEvent(Class<? extends SystemEvent>systemEventClass,
                             Class<?>sourceClass,
                             SystemEventListener listener)
```

其中，第一个参数指定注册的监听器要监听的系统事件的类型；第二个参数指明注册的监听器要监听的系统事件的事件源类型，可以取 null；第三个参数指定要注册的系统事件监听器。

组件系统事件源于某个 UIComponent 组件，UIComponent 对象的 subscribeToEvent 方法用于注册一个组件系统事件监听器，注册的监听器将监听由该 UIComponent 对象引发的系统事件。

```
public void subscribeToEvent(Class<? extends SystemEvent>eventClass,
                             ComponentSystemEventListener componentListener)
```

其中，第一个参数指定注册的监听器要监听的系统事件的类型；第二个参数指定要注册的组件系统事件监听器。

与其他类型的事件监听器一样，采用声明方式注册系统事件监听器是一种更为常用的方法。要注册一个普通系统事件监听器，可以在 JSF 配置文件中，用嵌套于 application 元素的 system-event-listener 子元素进行声明。

```
<application>
    <system-event-listener>
        <system-event-class>javax.faces.event.PostConstructApplicationEvent
            </system-event-class>
        <system-event-listener-class>listener.MySystemEventListener
            </system-event-listener-class>
```

```
    </system-event-listener>
</application>
```

其中，system-event-class 子元素指定被注册的监听器要监听的系统事件类的完整类名；system-event-listener-class 子元素指定被注册的系统事件监听器的类的完整类名。另外，可以提供可选的 source-class 子元素指明被注册的监听器要监听的系统事件的事件源类型。

在 application 元素中，可以嵌入多个 system-event-listener 元素，注册多个系统事件监听器。

可以采用声明方式注册一个独立的组件系统事件监听方法，具体做法是：在页面文件中，用嵌套于某组件标记的 f:event 子标记进行声明，被注册的监听方法将监听该组件引发的特定类型的组件系统事件。下面标记在视图根上注册一个监听方法，用于监听视图根引发的特定类型的事件。

```
<f:metadata>
    <f:event type="preRenderView" listener="#{myBean.method}"/>
</f:metadata>
```

其中，type 属性指定要监听的组件系统事件类型，可以是事件类的完整类名，也可以是它的短名(short name)。表 8-2 列出了一些组件系统事件类的短名。

表 8-2　组件系统事件类的短名

完 整 类 名	短　　名
javax.faces.event.PostAddToViewEvent	postAddToView
javax.faces.event.PreValidateEvent	preValidate
javax.faces.event.PostValidateEvent	postValidate
javax.faces.event.PreRenderViewEvent	preRenderView
javax.faces.event.PreRenderComponentEvent	preRenderComponent

f:event 标记的 listener 属性指定一个方法表达式，作为独立的组件系统事件监听方法。当所在的组件引发指定类型的事件时，该监听方法将被调用。独立的组件系统事件监听方法的方法签名应该如下：

```
public void <方法名> (ComponentSystemEvent event)
```

或

```
public void <方法名> ()
```

独立的组件系统事件监听方法可以抛出不受检查的 AbortProcessingException 例外，以便通知 JSF 框架：该事件不必做进一步的处理，即不需要再广播至其他的监听器。

8.5.4　系统事件应用示例

本小节以论坛应用的注册功能为例，介绍利用系统事件实现多组件验证的方法，同时也

介绍对一个组件值实现提前预验证的方法。

在实现注册功能时，除需确保相关输入项为非空外，还要验证用户名是否已用以及两次输入的密码是否一致。在之前的论坛应用实现中，后面两项验证任务都是在动作方法中完成的。现在，希望把这些验证工作提前到“处理验证”阶段、甚至“应用请求值”阶段进行。

验证用户两次输入的密码是否相同的处理逻辑是非常简单的，但常规的验证器却无法实现这一功能。因为每一个验证器只能对某一组件的值进行验证处理，而不能对多个组件的值进行综合验证。利用组件系统事件 postValidate 及相应的监听方法，可以较好地克服普通验证器的上述限制。

即时验证用户名是否合法，而不是等所有的输入项都输入后再一起验证，是一种好的做法。这里采用的技术与 8.4.3 小节中针对“部门名称”组件所使用的技术基本类似，即将组件的 onchange 属性值设置为 submit()，使得当用户改变组件值时，就会即时引起表单的提交和请求。详细情况后面再介绍。

首先创建一个名为 ch8_registry 的 JSF 应用项目，然后从应用项目 luntan_logandreg（在第 4.11 节介绍）中复制所有的 JSF 页面、源包中所有的 Java 包和 Java 类。虽然本项目只需要修改 registry.xhtml 页面文件和 bean.Registry 受管 bean 类，但包含其他文件便于运行和调试，比如在完成注册后，可以在主页中选择退出，然后再通过登录页面进行登录。该应用的运行效果如图 8-5 所示。



图 8-5　应用 ch8_registry 运行效果图

1. 注册页面

修改后的注册页面 registry.xhtml 见代码清单 8-6，其中省略了部分无修改且无关紧要的代码。

实际上，对页面文件的修改有两处。一处是在 h:form 标记下增加了一个 f:event 标记。该标记在表单组件上注册了一个 postValidate 事件监听方法。postValidate 是一种组件系统事件，其监听方法一般注册于命名容器组件，如表单组件、表格组件，或者注册于视图根组件。当该容器组件中的所有输入类组件都已完成验证处理后，该 postValidate 组件系统事件将引发，已注册的监听方法将被执行。可以看出，利用该事件机制可以实现对某容器组件内若干组件值的综合验证。但需要注意，postValidate 事件的引发并不取决于容器组

件内各组件是否都分别通过了验证。也就是说，不管各组件是否通过验证，postValidate 事件都将引发。在编写相应的事件处理方法时，需了解这一特点。

另一处是对“用户名”文本域标记的修改，其中添加了三个属性。这三个属性的设置都不陌生：

- onchange＝"submit()"：一旦用户修改组件值，提交表单。
- validator＝……：在组件上注册一个验证方法。
- immediate＝"true"：组件值的转换和验证发生于“应用请求值”阶段。

在这里，无论是因用户修改用户名引起的表单提交和请求，还是因用户单击“提交”按钮引起的表单提交和请求，JSF 框架都会在“应用请求值”阶段用指定的验证方法对用户名进行验证。但在两种情况下的处理要求却是不一样的：前者在处理完验证后返回源页面，若验证失败，应显示相应的错误消息；后者在处理完验证后，若验证失败，则返回源页面，显示相应错误信息，若验证成功，则应继续 JSF 请求处理生命周期，如进入“验证处理”阶段、对其他组件值进行转换和验证处理等。

代码清单 8-6　registry.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>轻松论坛--注册</title>
  </h:head>
  <h:body>
    <p>
      <h:outputText value="请输入你的信息:"/>
    </p>
    <h:form id="fi">
      <f:event type="postValidate" listener="#{registry.validatePassword}"/>
      用 户 名
      <h:inputText id="username" value="#{registry.client.username}"
          required="true" requiredMessage="用户名为必填项"
          size="10" maxlength="15"
          immediate="true"
          validator="#{registry.validateUsername}"
          onchange="submit()"/> *
      <h:message for="username"/>
      <br/>
      密　　码
      <h:inputSecret id="password" value="#{registry.client.password}"
          required="true" requiredMessage="密码为必填项"
          size="10" maxlength="15"/> *
```

```
        <h:message for="password"/>
        <br/>
        确认密码
        <h:inputSecret id="password1" value="#{registry.password1}"
            required="true" requiredMessage="密码为必填项"
            size="10" maxlength="15"/> *
        <h:message for="password1"/>
        <br/>
        ...           //省略部分代码
        <p>
          <h:commandButton type="reset" value="重置"/>
           
           
          <h:commandButton id="b" value="提交" action="#{registry.registry}"/>
        </p>
      </h:form>
    </h:body>
</html>
```

2. 受管 bean

修改后的受管 bean 类 bean. Registry 见代码清单 8-7,其中添加了两个方法、修改了一个方法。添加的 validateUsername 方法是注册在“用户名”文本域上的验证方法,用于验证用户名是否已被使用。添加的 validatePassword 方法是注册在表单组件上的 postValidate 事件监听方法,用于验证用户两次输入的密码是否相同。修改的是 registry 方法,是“提交”按钮的动作方法。上述验证功能原来都是由 registry 方法完成的,所谓对 registry 方法的修改就是从中删除实现这些验证功能的代码。

validateUsername 方法首先对用户名进行验证。如果验证失败,则往消息队列添加一个消息,然后等到当前阶段(“应用请求值”阶段)结束时,转入“呈现响应”阶段,呈现源页面。如果验证成功,则判断本次请求是否由用户单击“提交”按钮引起,如果不是,则与验证失败时的情况类似,转去呈现源页面;如果是,则继续 JSF 请求处理生命周期。

validatePassword 方法首先获得两个密码组件,然后判断它们是否有效,即两个密码组件是否通过了各自的验证。如果各自都通过了验证,表明它们是非空的,这时可以获取经过验证的本地值,并判断两个值是否相同。如果两个密码组件并非都有效,则表明它们之中至少有一个是验证失败的(为空值),这时就没有继续处理的必要了。

代码清单 8-7　bean. Registry

```
package bean;
import entity.Client;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
```

```
import javax.faces.event.ComponentSystemEvent;
import model.ClientManager;
import util.ELUtil;

@ManagedBean
@RequestScoped
public class Registry {
  private Client client=new Client();
  public Client getClient(){
    return client;
  }
  public void setClient(Client client){
    this.client=client;
  }
  private String password1;
  public String getPassword1(){
    return password1;
  }
  public void setPassword1(String password1){
    this.password1=password1;
  }

  public void validateUsername(FacesContext context,UIComponent component,Object value){
    ClientManager cm=new ClientManager();
    Client client1=cm.findClientByName((String)value);
    if(client1!=null){
      FacesMessage msg=new FacesMessage("用户名已被使用");
      FacesContext.getCurrentInstance().addMessage(component.getClientId(),msg);{
      FacesContext.getCurrentInstance().renderResponse();
    }
    Object ok=util.ELUtil.evalEL("#{param['fi:b']}");
    if(ok==null){
      FacesContext.getCurrentInstance().renderResponse();
    }
  }
  public void validatePassword(ComponentSystemEvent cse){
    UIComponent source=cse.getComponent();
    UIInput pass=(UIInput) source.findComponent("fi:password");
    UIInput pass1=(UIInput) source.findComponent("fi:password1");
    if(pass.isValid()&&pass1.isValid()){
      String spass=(String)pass.getLocalValue();
      String spass1=(String)pass1.getLocalValue();
      if(!spass1.equals(spass)){
        FacesMessage msg=new FacesMessage("两次输入的密码不一致");
        FacesContext.getCurrentInstance().addMessage("fi:password1",msg);
```

```
        FacesContext.getCurrentInstance().renderResponse();
      }
    }
  }
  public String registry(){          //注册“提交”动作方法
    ClientManager cm=new ClientManager();
    client.setStatus('1');
    boolean f=cm.insertClient(client);
    if(!f){
      FacesMessage msg=new FacesMessage("注册出错");
      FacesContext.getCurrentInstance().addMessage("fi:username",msg);
      return null;
    }
    SessionInfo sessinfo=(SessionInfo)ELUtil.getBean("sessinfo");
    sessinfo.setClient(client);
    return "index";
  }
}
```

8.6 小　结

- JSF 事件包括 Faces 事件、阶段事件和系统事件三大类，Faces 事件又分动作事件和值变化事件两种。
- 动作事件（ActionEvent）的事件源是动作类组件，其监听器接口是 ActionListener。
- 值变化事件（ValueChangeEvent）的事件源包括基本输入类组件、选择类组件和视图参数组件等，其监听器接口是 ValueChangeListener。
- 阶段事件（PhaseEvent）的事件源是 Lifecycle 对象，其监听器接口是 PhaseListener。
- 系统事件（SystemEvent）是一类事件的总称。一种系统事件的事件源可以是某种任意类型对象。任何一种系统事件的监听器接口都是 SystemEventListener。
- 组件系统事件（ComponentSystemEvent）是系统事件的子类型，也是一类事件的总称。一种组件系统事件的事件源可以是某种 UIComponent 型对象。任何一种组件系统事件的监听器接口都是 ComponentSystemEventListener。
- 事件监听器在能够监听事件之前，必须先进行注册。各种事件监听器有相应的注册方法。

习　题　8

1. 术语解释。
- 事件源；
- 事件对象；
- 事件监听器。

2．动作事件是如何产生的？一个动作事件是否可以有多个监听器或监听方法？这些监听器和监听方法按何种顺序被通告或调用？

3．如何定义值变化监听器类？如何注册一个值变化监听器？

4．如何定义阶段监听器类？如何注册阶段监听器？

5．简述注册系统事件监听器和组件系统事件监听器的方法。

6．比较值变化事件与 postValidate 组件系统事件发生的条件、时间。

7．修改应用项目 sh4_logandreg（第 4 章第 5 题）中的注册功能：能对用户名进行提前预验证；能在“处理验证”阶段结束前，对用户两次输入的密码的一致性进行验证。

第 9 章　资源包与国际化

本章主题：

- 创建资源包
- 资源包类与属性文件
- 在 JSF 中使用资源包
- 场所(Locale)
- 国际化

资源包(Resource Bundle)是 Java 标准版提供的特性，并在 Java 企业版中得到进一步的支持。在 JSF 中使用资源包还有其自己的特点，如通过注册声明后就可以在 EL 值表达式中直接访问资源包，而不需要用代码显式创建或获取资源包。

支持应用软件国际化是资源包固有的功能。在 JSF 应用中，通过使用资源包和消息包，不仅可以将 JSF 应用中各页面的静态文本等从页面中分离出来，使它们能被集中管理和重用，而且可使 JSF 应用能够向来自不同国家、使用不同语言的用户呈现符合他们阅读习惯的响应。

9.1　创建资源包

资源包是 ResourceBundle 抽象类的某个具体子类的实例，由一组键一值对组成。键总是字符串(区分大小写)，值可以是任意类型的对象。在很多情况下，值也是字符串。

ResourceBundle 抽象类的某个具体子类也称为资源包类。要创建资源包，一般应先定义资源包类，然后再用特殊的方法创建或获取资源包。

9.1.1　扩展 ResourceBundle 类

可以扩展 ResourceBundle 抽象类，定义资源包类。ResourceBundle 的具体子类必须覆盖以下两个抽象方法：

(1) protected Object handleGetObject(String key)

返回当前资源包中指定键的资源。若不存在相应的资源，返回 null。

(2) public Enumeration<String> getKeys()

返回前资源包中所有资源的键的枚举对象。

BundleOne 是一个简单的 ResourceBundle 具体子类的示例。它包含了三个 String 型的资源，其中第 3 个资源是一个包含参数的消息模板，所有资源保存在一个 Map 对象中。见代码清单 9-1。

代码清单 9-1　资源包类 bundle.BundleOne

```
1. package bundle;
```

```java
import java.util.Collections;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.Map;
import java.util.ResourceBundle;

public class BundleOne extends ResourceBundle{
  private Map<String,String>resources=new HashMap<String,String>();
  public BundleOne(){
    resources.put("prompt","请输入:");
    resources.put("button_title","确认");
    resources.put("result","你输入的值是:{0}");
  } }
  @Override
  protected Object handleGetObject(String key){
    return resources.get(key);
  }
  @Override
  public Enumeration<String>getKeys(){
    return Collections.enumeration(resources.keySet());
  }
}
```

9.1.2　扩展 ListResourceBundle 类

定义资源包类的一种更为方便的方法是扩展 ListResourceBundle 类，定义一个具体的子类。ListResourceBundle 是 ResourceBundle 类的抽象子类。ListResourceBundle 的具体子类必须覆盖以下抽象方法：

```java
protected Object[][] getContents()
```

该方法返回当前资源包中所有资源的键－值对。方法返回一个 Object[][]型对象，其中每一个内层数组由两个元素组成，第 1 个元素表示一个资源的键，必须是字符串，第 2 个元素表示该资源的值。

BundleTwo 是一个简单的 ListResourceBundle 具体子类的示例。它定义了与 BundleOne 类一样的资源。见代码清单 9-2。

代码清单 9-2　资源包类 bundle.BundleTwo

```java
package bundle;
import java.util.ListResourceBundle;

public class BundleTwo extends ListResourceBundle {
  @Override
  protected Object[][] getContents(){
    return new Object[][] {
      {"prompt","请输入:"},
```

```
        {"button_title","确认"},
        {"result","你输入的值是:{0}"} };
    }
}
```

9.1.3 资源包的获取与使用

资源包是资源包类的实例，但资源包并不是简单地利用 new 表达式来创建的，而是通过调用 ResourceBundle 类的 getBundle 方法来创建或获取。

getBundle 是 ResourceBundle 类中定义的一组重载的静态方法，用于创建或获取一个指定的资源包。下面是 getBundle 方法的一种基本格式：

```
public static final ResourceBundle getBundle(String baseName)
```

该方法创建指定基名的资源包，基名是相应资源包类的完整类名。默认情况下，创建的资源包将被缓存，下次再调用该方法时将直接返回已存在的资源包。若不存在指定的资源包，方法抛出不受检查的 java.util.MissingResourceException 例外。

一旦获得了资源包，就可以通过下面方法获取资源包中指定键的资源。

```
public final Object getObject(String key)
```

如果能确保资源是 String 型的，那么也可以用下面方法获取指定键的资源。

```
public final String getString(String key)
```

与 getBundle 方法类似，对于 getObject 或 getString 方法，如果当前资源包中不存在指定键的资源，那么也会抛出 MissingResourceException 例外。

第 4.4 节曾介绍如何用 h:outputFormat 和 f:param 标记在 JSF 页面中参数化和显示一个消息模板。在 Java 代码中，则可以利用 java.text.MessageFormat 类的 format 静态方法实现消息模板的参数化。

```
public static String format(String pattern,Object... arguments)
```

其中，方法的第 1 个参数指定消息模板，后面可以根据消息模板中的参数数目指定相应数目的参数值。

代码清单 9-3 是一个 Java 应用程序，演示了如何获取资源包和使用资源包。程序首先获取基名为 bundle.BundleOne 的资源包，然后输出了该资源包中所有资源的键和值，最后参数化并输出键为“result”的消息模板资源。

代码清单 9-3　资源包的获取与使用

```
import java.text.MessageFormat;
import java.util.Enumeration;
import java.util.ResourceBundle;

public class BundleDemo {
  public static void main(String[] args){
    ResourceBundle rb=ResourceBundle.getBundle("bundle.BundleOne");
```

```
        Enumeration<String>keys=rb.getKeys();
        while(keys.hasMoreElements()){
          String key=keys.nextElement();
          System.out.println(key+":"+rb.getString(key));
        }
        String r_result=rb.getString("result");
        String msg=MessageFormat.format(r_result,"China");
        System.out.println(msg);
    }
}
```

下面是程序的运行结果：

```
result:你输入的值是:{0}
prompt:请输入:
button_title:确认
你输入的值是:China
```

说明：资源包类、资源包与普通的类、对象相比有明显的不同，基于某个资源包类创建的各资源包对象总是具有相同的资源，这些资源定义在资源包类中。所以有时也把资源包类直接称为资源包。

9.1.4 PropertyResourceBundle 类与属性文件

如果一个资源包的资源都是字符串文本或消息模板，那么也可以把它们定义在一个属性文件中。属性文件是一个扩展名为.properties 的文本文件，包含一系列属性定义。在属性文件中，每个属性定义占一行，包括属性名和属性值。

```
<属性名>=<属性值>
```

当把一个属性文件看作是一个资源集时，文件中的每个属性就是资源包中的一个资源。其中，属性名是资源的键，属性值就是资源的值。属性文件 BundleThree.properties 定义了与资源包类 BundleOne 和 BundleTwo 一样的资源，见代码清单 9-4。

代码清单 9-4　属性文件 bundle/BundleThree.properties

```
prompt=请输入:
button_title=确认
result=你输入的值是:{0}
```

从用户的角度来看，属性文件的作用类似于资源包类。属性文件应该与资源包类存放在相同的位置，比如属性文件 BundleThree.properties 与资源包类 BundleOne 和 BundleTwo 都存放在 bundle 目录下。利用 ResourceBundle 类的 getBundle 静态方法同样可以基于属性文件创建或获取一个资源包。下面代码通过指定属性文件 BundleThree.properties 创建或获取相应的资源包：

```
ResourceBundle rb=ResourceBundle.getBundle("bundle.BundleThree");
```

getBundle 方法执行时，首先会查看在缓存中是否已有指定的资源包。若存在相应的

资源包，那么直接返回即可。若不存在相应的资源包，则查找是否存在指定的资源包类。若存在相应的资源包类，则实例化、缓存并返回。如果不存在资源包类，则查找是否存在相应的属性文件（基名中的“.”用“/”替换，并加扩展名.properties）。若存在相应的属性文件，则创建资源包类 PropertyResourceBundle 的一个实例，并与相应的属性文件建立关联，然后缓存、返回该资源包实例。

PropertyResourceBundle 是 ResourceBundle 类的一个具体子类。与其他资源包类不同，该资源包类本身并没有定义任何资源。在创建 PropertyResourceBundle 型资源包实例时，需要将其与某个属性文件建立关联，它把属性文件的内容当作自己所管理的资源。

说明：由于基于某个属性文件创建的各 PropertyResourceBundle 型资源包实例都具有相同的资源，这些资源定义在属性文件中，所以有时也把这些包含资源的属性文件直接称为资源包。

9.2 在 JSF 中使用资源包

JSF 提供对 Java 资源包的支持。在 JSF 中，可以通过声明的方式创建资源包，然后利用 EL 表达式访问资源。消息包是一种特殊的资源包，在 JSF 中起着重要的作用，本节也将作详细介绍。

9.2.1 资源包的注册、装入与使用

在 JSF 应用中，要使用资源包，同样需要先定义资源包类或属性文件。这些资源包类或属性文件通常定义于源包。比如，可以将资源包类 BundleOne.java、BundleTwo.java 和属性文件 BundleThree.properties 都存放在源包的 bundle 包或目录中。

在 JSF 应用中，一般不需要通过 ResourceBundle 类的 getBundle 方法显式创建或获取资源包，而只需通过声明告诉 JSF 框架该应用或该页面需要使用什么类型的资源包，JSF 框架会在需要时自动创建相应的资源包并进行管理和维护。

可以在 Faces 配置文件中用 resource-bundle 元素进行声明，其中 base-name 子元素指定资源包的基名，var 子元素暴露资源包实例的名称。

代码清单 9-5 注册了一个基名为 bundle.BundleOne 的资源包。在当前应用的 EL 值表达式中，可以通过 var 子元素暴露的名称 rb1 访问该资源包中的资源，如 #{rb1.prompt}。

代码清单 9-5 注册资源包

```
<application>
  <resource-bundle>
    <base-name>bundle.BundleOne</base-name>
    <var>rb1</var>
  </resource-bundle>
  ......
</application>
```

这种声明的方式也称为注册资源包。通过这种方式注册的资源包只实例化一次，在各

页面中都可被使用。

也可以在 JSF 页面中用 f:loadBundle 标记进行声明，其中 basename 属性指定资源包的基名，var 属性暴露资源包的名称。

下面标记可以放在 JSF 页面中，用于装入了一个基名为 bundle.BundleThree 的资源包，即基于属性文件 bundle/BundleThree.properties 的资源包。在当前页面的 EL 值表达式中，可以通过 var 属性暴露的名称 rb3 访问该资源包中的资源，如 #{rb3.button_title}。

```
<f:loadBundle basename="bundle.BundleThree" var="rb3"/>
```

这种声明的方式也称为装入资源包。通过这种方式装入的资源包只能在当前页面被使用，其作用域是请求范围的。

显而易见，注册资源包比装入资源包更为有效，因为每个注册资源包在整个应用作用域内只需创建一次。

在一个 JSF 应用中，可以注册或装入多个资源包，每个资源包可以通过 var 子元素或属性暴露的资源包名称进行访问。

9.2.2 资源包应用示例

该应用项目(ch9_bundle)演示资源包的应用，主要由一个资源包类、一个受管 bean 和一个 JSF 页面组成。JSF 页面的呈现效果如图 9-1 所示。



图 9-1 应用 ch9_bundle(使用资源包)

1. 资源包

首先在应用项目的源包下创建一个名为 bundle 的 Java 包，并在该包中创建资源包类 BundleOne.java，具体代码见代码清单 9-1。

然后再为应用项目在 WEB-INF 下创建 Faces 配置文件 faces-config.xml，并在其中注册上述资源包。具体代码见代码清单 9-5。

2. 受管 bean

该应用包含一个受管 bean，文件名为 Index.java(代码清单 9-6)。其中定义了一个可读写的 String 型的属性 value。

代码清单 9-6 受管 bean(Index.java)

```
package bean;
```

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class Index {
  private String value="";
  public String getValue(){
    return value;
  }
  public void setValue(String value){
    this.value=value;
  }
}
```

3. JSF 页面

代码清单 9-7 给出的 JSF 页面演示了资源包的使用。其中文本域的标签，递交按钮的标题以及按钮下方的响应文本都来自资源包。

代码清单 9-7　JSF 页面(index.xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>ResourceBundleDemo</title>
  </h:head>
  <h:body>
    <h:form id="f">
      <p><h:outputLabel for="i" value="#{rb1.prompt}"/></p>
      <p><h:inputText id="i" value="#{index.value}"/></p>
      <p><h:commandButton value="#{rb1.button_title}"/></p>
    </h:form>
    <p>
    <h:outputFormat value="#{rb1.result}">
      <f:param value="#{index.value}"/>
    </h:outputFormat>
    </p>
  </h:body>
</html>
```

9.2.3　消息包及其使用

消息包是资源包特性在 JSF 中的另一个具体应用。在 JSF 中，当转换或验证出错时，

应产生一个 FacesMessage 消息、并抛出例外，这些消息会被自动放入消息队列。另外，在处理各类事件时，也可以产生 FacesMessage 消息，并将其放入消息队列。当页面再次呈现时，消息队列中的消息将通过 h:message 或 h:messages 组件被显示出来。

一个 FacesMessage 消息由概要文本、详细文本和严重级别三部分组成。其中概要文本和详细文本可以由消息包管理。消息包是一种特殊的资源包，其中每一个资源的键称为消息 ID，每一个资源的值总是 String 型的，称为消息文本（包括消息模板）。由于一个消息既有概要文本，又有详细文本，所以一个消息可能包括两个资源，一个指定消息的概要文本，一个指定消息的详细文本。这里，指定详细文本的消息 ID 总是以_detail 结尾，而其前面部分则与相应的指定概要文本的消息 ID 相同。

与一般的资源包一样，消息包也应该基于资源包类或者基于属性文件来创建。由于消息资源总是 String 型的，所以更多的是采用属性文件的形式。下面是用作消息包的属性文件的内容的一般格式：

```
msg1=…
msg1_detail=…{0}…
```

这里仅定义了一个消息的概要文本和详细文本，其中详细文本包含一个参数。一般来说，无论是概要文本还是详细文本，都可以是普通的消息文本，也可以是带参数的消息模板。

在 NetBeans IDE 中，创建一个属性文件的一般步骤如下（在“项目”窗口中进行）：

(1) 单击选择项目结点。

(2) 从“文件”菜单，选择“新建文件”命令，打开“新建文件”对话框。

(3) 选择文件类型：其他|属性文件。

(4) 指定文件名（不需要指定扩展名），并指定存放属性文件的文件夹。

(5) 单击“完成”按钮。

要使用消息包，首先需要在 Faces 配置文件中用 message-bundle 元素进行声明和注册，其中 message-bundle 元素是 application 元素的子元素，用于指定消息包的基名。

```
<message-bundle>…</message-bundle>
```

与一般的资源包不同，一个 JSF 应用只能注册一个消息包，该消息包称为由用户提供的自定义消息包。相对应的，框架本身包含的消息包（基名为 javax.faces.Messages）则称为由应用提供的标准消息包。

注册了消息包，就可以访问消息包，获取其中的消息文本，进而创建消息对象。下面介绍使用消息包的一般过程和方法。

(1) 获取消息包的基名

调用 Application 对象的 getMessageBundle 方法可以返回应用中用户提供的自定义消息包的基名。

```
FacesContext context=…;
Application app=context.getApplication();
String baseName=app.getMessageBundle();
```

一些转换器、验证器需要被设计成是可重用的，这时把消息包的基名硬编码到代码中是

不行的。如果用户在应用中没有注册消息包，那么 getMessageBundle 方法返回 null。

(2) 根据基名获取消息包

消息包也是资源包，可以调用 ResourceBundle 类的 getBundle 获取消息包。若不存在指定的消息包，getBundle 方法将抛出 MissingResourceException 例外。

```
ResourceBundle mb=ResourceBundle.getBundle(baseName);
```

(3) 根据消息 ID 获取消息文本

与从资源包中获取资源一样，可以调用消息包的 getString 方法获取指定消息 ID 的消息文本。若不存在指定的消息文本，getString 方法将抛出 MissingResourceException 例外。

```
String mid="msg1";
String summary=mb.getString(mid);
String detail=mb.getString(mid+"_detail");
```

(4) 参数化消息模板

如果消息文本是消息模板，则可以调用 MessageFormat 类的 format 方法对消息模板进行格式化操作，产生参数化的消息文本。

```
Object value=…                //参数
detail=MessageFormat.format(detail,value);
```

(5) 创建消息对象

利用消息文本创建 FacesMessage 类的实例，并指定消息的严重级别。

```
FacesMessage fmsg=new FacesMessage(summary,detail);
fmsg.setSeverity(FacesMessage.SEVERITY_ERROR);
```

(6) 将消息对象加入消息队列

在请求处理过程中，如果出现例外情况，应该将产生的消息对象放入消息队列，以便在页面重新呈现时能显示出这些错误消息。

对于转换器或验证器，在处理过程中如果出现例外情况，应该以产生的消息对象为参数创建相应的例外对象并抛出。这样，消息对象会被自动放入消息队列。

在处理各类事件时，如果出现例外情况，可以调用 FacesContext 型对象的 addMessage 方法将消息对象加入消息队列。

```
FacesContext context=…
context.addMessage("f:i",fmsg);
```

其中，第 1 个参数指定某组件的客户端 ID，作为该消息的源。若该参数为 null，表明添加的是一个全局消息。第 2 个参数指定一个 FacesMessage 对象。

在上述一系列步骤中，(1)至(5)步的功能实质上是利用消息 ID 和格式化消息模板所需的参数创建一个 FacesMessage 对象。虽然 JSF 规范没有提供相应的编程接口，但其参考实现却包含一个能完成此功能的方法，即 MessageFactory 类的 getMessage 方法。在有些场合可以考虑使用该方法来直接创建 FacesMessage 对象。

```
public static FacesMessage getMessage(String messageId,Object... params);
```

其中 MessageFactory 类定义于 com. sun. faces. util 包。

9.2.4 替换标准消息文本

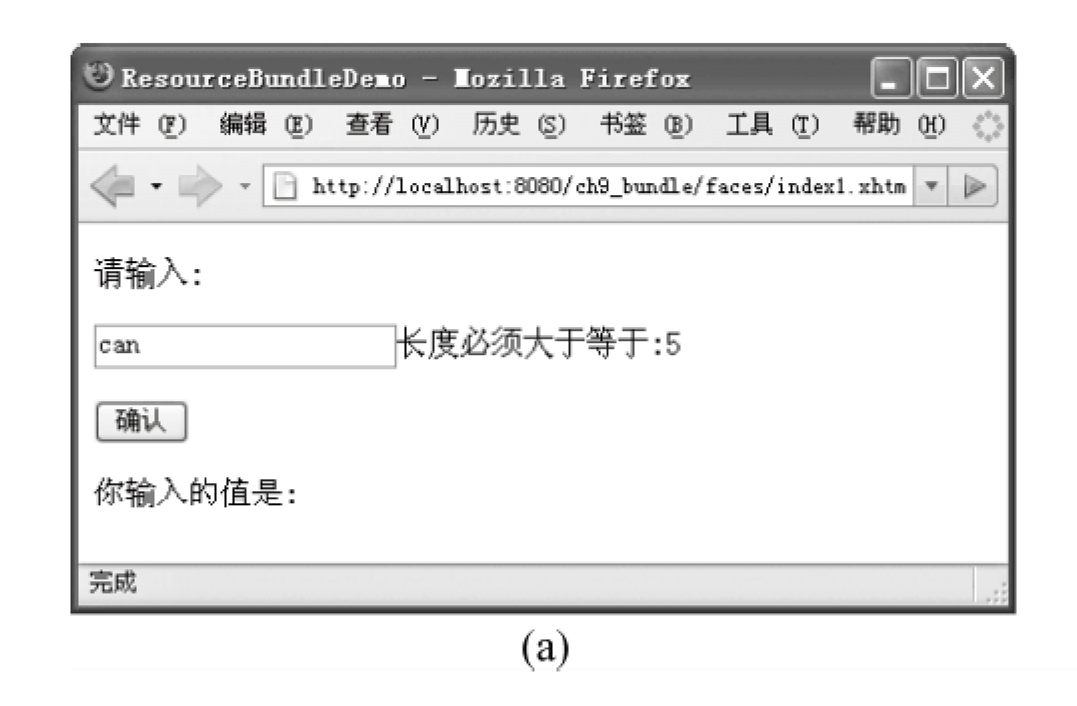
第 7 章的表 7-3 和表 7-6 列出了一些常用的标准转换消息和验证消息，这些消息文本都定义在 JSF 框架提供的标准消息包 javax. faces. Messages 中。当出现异常情况时，标准转换器和验证器通常利用这些消息文本创建 FacesMessage 对象并抛出例外。

可以由用户自定义的消息文本替换标准的消息文本，使得标准转换器和验证器能根据这些自定义的消息文本来创建消息对象。替换标准消息文本，通常并不是指修改标准消息包中的相关消息文本，而是指在自定义消息包中定义与某标准消息具有相同消息 ID 的消息文本。当出现异常情况时，标准转换器和验证器会先从自定义消息包中寻找具有指定消息 ID 的消息文本。如果在自定义消息包中找不到所需的消息文本，再从标准消息包中寻找具有指定消息 ID 的消息文本。然后利用消息文本创建消息对象。

在 JSF 规范的参考实现中，标准转换器和验证器实际上都是调用 MessageFactory 类的 getMessage 方法来完成上述功能的。也就是说，getMessage 方法会依次从自定义消息包、标准消息包寻找指定消息 ID 的消息文本，然后创建一个 FacesMessage 对象返回。

9.2.5 消息包应用示例

该示例继续 9.2.2 节介绍的应用项目(ch9_bundle)，用于演示消息包的使用。在项目中增加了一个消息包、一个自定义验证器和一个 JSF 页面。JSF 页面的呈现效果如图 9-2 所示，其中：(a)显示了标准验证器根据自定义消息文本产生的错误消息，(b)显示了自定义验证器产生的错误消息。



(a)



(b)

图 9-2 应用 ch9_bundle(使用消息包)

1. 消息包

首先在 bundle 包下创建属性文件 mymsg. properties，见代码清单 9-8。其中定义了一个消息(消息 ID 为 msg1)的概要文本和详细文本。另外重新定义了一个标准验证消息文本，该消息文本会被标准验证器 LengthValidator 使用。

代码清单 9-8 属性文件(mymsg. properties)

```
1. msg1=包含非字母字符
2. msg1_detail=包含非字母字符：{0}
```

```
javax.faces.validator.LengthValidator.MINIMUM=长度必须大于等于:{0}
```

然后在 Faces 配置文件 faces-config.xml 中注册该消息包，见代码清单 9-9。

代码清单 9-9　注册消息包

```
<application>
  <message-bundle>bundle.mymsg</message-bundle>
  ......
</application>
```

2. 自定义验证器

首先在应用项目的源包下创建一个 Java 包 validator，然后在该包中创建一个自定义验证器类 MyValidator.java（代码清单 9-10）。该验证器用于验证用户从文本域中输入的字符串是否仅包含字母。如果包含非字母字符，验证器将从消息包中获取相应的消息文本，然后创建 FacesMessage 对象，并抛出例外。

代码清单 9-10　自定义验证器类（MyValidator.java）

```
package validator;
import java.text.MessageFormat;
import java.util.MissingResourceException;
import java.util.ResourceBundle;
import javax.faces.application.Application;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.FacesValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

@FacesValidator("myvalidator")
public class MyValidator implements Validator {
  @Override
  public void validate(FacesContext context,UIComponent component,Object value)
                                  throws ValidatorException {
    StringBuilder sb=new StringBuilder(value.toString());
    int i=0;
    while(i<sb.length()){
      char ch=sb.charAt(i);
      if(!Character.isLetter(ch)){
        Application app=context.getApplication();
        String baseName=app.getMessageBundle();
        if(baseName!=null){
          String mid="msg1";
          String summary=null;
          String detail=null;
          try {
```

```
            ResourceBundle mb=ResourceBundle.getBundle(baseName);
            summary=mb.getString(mid);
            detail=mb.getString(mid+"_detail");
          } catch(MissingResourceException mre){}
          detail=MessageFormat.format(detail,value);
          FacesMessage fmsg=new FacesMessage(summary,detail);
          fmsg.setSeverity(FacesMessage.SEVERITY_ERROR);
          throw new ValidatorException(fmsg);
        }
      }
      i++;
    }
  }
}
```

类头的标注@FacesValidator("myvalidator")表明该自定义验证器类按标注方式进行了注册，注册 ID 为 myvalidator。

3. JSF 页面

创建 JSF 页面 index1.xhtml(代码清单 9-11)。该页面与 9.2.2 节介绍的页面 index.xhtml 基本相同，只是为文本域引用了两个验证器：一个是标准验证器，确保输入串的长度必须大于等于 5；一个是自定义验证器，要求用户只能输入字母。另外为文本域设置了一个消息组件，用于显示验证出错消息。

代码清单 9-11　JSF 页面(index1.xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>ResourceBundleDemo</title>
  </h:head>
  <h:body>
    <h:form id="f">
      <p><h:outputLabel for="i" value="#{rb1.prompt}"/></p>
      <p>
      <h:inputText id="i" value="#{index.value}">
        <f:validateLength minimum="5"/>
        <f:validator validatorId="myvalidator"/>
      </h:inputText>
      <h:message for="i"/>
      </p>
      <p><h:commandButton value="#{rb1.button_title}"/></p>
    </h:form>
```

```
    <p>
    <h:outputFormat value="#{rb1.result}">
      <f:param value="#{index.value}"/>
    </h:outputFormat>
    </p>
  </h:body>
</html>
```

9.3 国 际 化

一个场所代表一个有特定地理、政治或文化的区域，每个场所有各自的语言、传统或习惯。国际化(Internalization，i18n)是指通过设计，使软件能够适应多种场所的能力或潜力。本地化(Localization，l10n)是指在软件中添加与特定场所有关的信息使其能适应特定场所的过程。

就如上节所介绍的，使用资源包的一个好处是支持页面中的静态文本、消息以及消息模板等能从页面中分离出来集中管理，便于系统维护。使用资源包的另一个好处是支持国际化和本地化，即若要让应用适应一个新的场所，只需要添加额外的资源包或消息包，而不需要修改页面代码。本节介绍这方面的技术。

9.3.1 场所

在 Java 中，一个场所(Locale)用一个 Locale 类的实例表示，其状态主要由语言和国家(地区)两个数据表示。下面是 Locale 类的两个常用的构造方法：

```
Locale(String language)
Locale(String language,String country)
```

其中，参数 language 指定场所的语言，用 ISO-639 定义的、小写的两字母代码表示，如 en(英语)、zh(中文)等；参数 country 指定场所的国家或地区，用 ISO-3166 定义的、大写的两字母代码表示，如 CN(中国)、US(美国)等。在有些场合，也会用这些代码来表示某个场所，如 zh、zh_CN、en、en_US 等。

创建 Locale 实例时，构造方法会自动完成字母的大小写转换，即将语言代码转换成小写、将国家(地区)代码转换成大写，但不会对参数做进一步的验证。所以，若指定的参数不对，创建的 Locale 对象并不能表示一个有效的场所。但指定的参数必须是非 null，否则会抛出 NullPointerException 例外。

Locale 类定义了一些常用的 Locale 型的类的有名常量。很多时候，可以直接引用它们，而不必再显式创建表示相同场所的 Locale 实例。下面列出其中的几个，并对其状态值进行说明。

- Locale.ENGLISH：en(英语)。
- Locale.US：en_US(英语_美国)。
- Locale.UK：en_GB(英语_英国)。
- Locale.CHINESE：zh(中文)。

- Locale.CHINA：zh_CN(中文_中国)。
- Locale.TAIWAN：zh_TW(中文_台湾)。

Locale 类同样定义了一些公共的实例方法和类方法，下面是其中的几个。

(1) String getCountry()：返回当前场所的国家(地区)代码。

(2) String getLanguage()：返回当前场所的语言代码。

(3) getDisplayCountry()：返回当前场所的国家(地区)名称，该名称通常会依据默认场所进行本地化。

(4) getDisplayLanguage()：返回当前场所的语言名称，该名称通常会依据默认场所进行本地化。

(5) static Locale getDefault()：返回当前 Java 虚拟机的默认场所。

(6) static void setDefault(Locale newLocale)：设置当前 Java 虚拟机的默认场所。

LocaleDemo.java(代码清单 9-12)是一个 Java 应用程序，演示了 Locale 类中的一些常量和方法的使用。在不同的默认场所情况下，一个场所的语言、国家代码总是相同的，但其名称往往是不同的。

代码清单 9-12 使用 Locale 类

```
import java.util.Locale;
public class LocaleDemo {
  public static void main(String[] args){
    Locale locale=new Locale("en","US");
    System.out.println("Default:"+Locale.getDefault());
    System.out.println(locale.getLanguage()+"_"+locale.getCountry());
    System.out.println(locale.getDisplayLanguage()+"_"+locale.getDisplayCountry());
    Locale.setDefault(Locale.US);           //设置默认场所
    System.out.println("Default:"+Locale.getDefault());
    System.out.println(locale.getLanguage()+"_"+locale.getCountry());
    System.out.println(locale.getDisplayLanguage()+"_"+locale.getDisplayCountry());
  }
}
```

程序的运行结果如下：

```
Default:zh_CN
en_US
英文_美国
Default:en_US
en_US
English_United States
```

9.3.2 创建不同场所的资源包

对一组资源，应用可以为其所支持的各场所创建各自的资源包类(或属性文件)，这些资源包类形成一个资源包族。通常，一个资源包族应包含一个默认资源包类，该默认资源包类的类名称为资源包族的基名。资源包族中其他资源包类的类名应在基名的基础上添加它所

支持的场所的相关代码。

下面是基名为 bundle.BundleOne 的一个资源包族：

- bundle.BundleOne。
- bundle.BundleOne_zh。
- bundle.BundleOne_zh_CN。
- bundle.BundleOne_zh_TW。
- bundle.BundleOne_en。
- bundle.BundleOne_en_US。
- bundle.BundleOne_en_UK。

以上资源包族的每个成员既可以是 Java 类，也可以是属性文件。如果两者同时出现，优先使用类文件，属性文件被忽略。

总的来说，资源包族的每个成员定义有相同的资源条目，但每个资源条目在各成员中会有不同的值。但不能排除有些资源条目在不同的成员(特别是具有相同语言的各成员)中会有相同的值。所以进一步地，可以在只涉及语言的成员(如 bundle.BundleOne_en)中定义一些公共的资源条目，而在涉及国家(地区)的成员(如 bundle.BundleOne_en_US)中定义针对该国家(地区)特有的资源条目。

9.3.3 资源包链与资源定位

第 9.1.3 节曾经介绍 ResourceBundle 类的 getBundle 方法，它可以创建或获取指定基名的资源包。但基名究竟是什么？方法是否一定返回指定类的资源包实例？实际上还没有明确的说明。下面将再次列出该方法的两种格式，并做进一步的说明和讨论。

格式 1：

```
public static final ResourceBundle getBundle(String baseName)
```

格式 2：

```
public static final ResourceBundle getBundle(String baseName,Locale locale)
```

两种格式都需要指定基名。基名就是一个资源包族的基名，也即资源包族中默认资源包类的完整类名。getBundle 方法就是在指定的资源包族中创建或获取相关的资源包。

getBundle 方法是与场所相关的。如果没有指定场所参数(第 1 种格式)，方法创建或获取与 Java 虚拟机默认场所相关的资源包。以第 9.3.2 节所述资源包为例，如果 Java 虚拟机的默认场所为 zh_CN，那么下面代码：

```
ResourceBundle rb1=ResourceBundle.getBundle("bundle.BundleOne");
```

将创建或获取以下资源包，这些资源包形成一条链，称为资源包链。其中 BundleOne_zh 是 BundleOne_zh_CN 的父资源包，BundleOne 是 BundleOne_zh 父资源包。方法返回的资源包是 BundleOne_zh_CN。

- bundle.BundleOne_zh_CN。
- bundle.BundleOne_zh。
- bundle.BundleOne。

当调用 getBundle 方法返回的资源包 rb1 的 getString 或 getObject 方法时，首先会在当前资源包(BundleOne_zh_CN)中寻找指定的资源，如果在该资源包中找不到指定的资源，则方法自动到它的父资源包(BundleOne_zh)中寻找指定的资源，以此类推。

如果在调用 getBundle 方法时指定场所参数(第 2 种格式)，方法将创建或获取与指定场所以及 Java 虚拟机默认场所相关的资源。以 9.3.2 节所述资源包为例，如果默认场所为 zh_CN，那么下面代码：

```
ResourceBundle rb=ResourceBundle.getBundle("bundle.BundleOne", Locale.US);
```

将创建或获取以下资源包，这些资源包形成一条链，后面的资源包是前面资源包的父资源包。方法返回的资源包是 BundleOne_en_US。

- bundle.BundleOne_en_US。
- bundle.BundleOne_en。
- bundle.BundleOne_zh_CN。
- bundle.BundleOne_zh。
- bundle.BundleOne。

当调用 getBundle 方法返回的资源包 rb2 的 getString 或 getObject 方法时，首先会在当前资源包(BundleOne_en_US)中寻找指定的资源，如果在该资源包中找不到指定的资源，则方法自动到它的父资源包(BundleOne_en)中寻找指定的资源，以此类推。

通常，一个资源包族应包含一个默认资源包类。从上面的介绍可以看出，相应的默认资源包是用户定位资源的最后机会，也就是说，如果在其他相关资源包中找不到所需资源，最终都会到默认资源包中寻找。

但这并不是说，默认资源包是必不可少的，或者说，在一个资源包族中，默认资源包类是必须存在的。应该说，一个资源包族可以没有所谓的默认资源包类，但即使没有默认资源包类，资源包族的基名仍然是存在的，ResourceBundle 的 getBundle 仍然是根据基名来获取相关的资源包，或者说获取一个资源包链。当然，此时获得的资源包链中不会有默认资源包。

9.3.4 JSF 应用国际化

与独立的 Java 应用程序相比，JSF 应用的国际化强调的是它能被不同场所的用户访问，而不是一定要将它移植到不同的场所运行。当处于不同场所的用户通过互联网进行访问时，JSF 应用应该能用最合适的语言、习惯格式等产生用户能够理解和接受的响应。

考虑资源包在 JSF 应用中的国际化应用，首先应该为 JSF 应用设置合适的资源包族，为每一个资源包族创建为支持某些场所而所需的成员，相关概念与技术已在前面介绍。同时也需要向 JSF 框架声明该应用支持哪些场所以及它的默认场所是什么。这种声明是在 Faces 配置文件中使用 locale-config 元素来完成的，下面是一个示例。

```
<application>
   <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>en_US</supported-locale>
      <supported-locale>en</supported-locale>
```

```
        <supported-locale>zh_CN</supported-locale>
        <supported-locale>zh</supported-locale>
        ……
    </locale-config>
</application>
```

locale-config 元素是 application 元素的子元素，其中可包含一个 default-locale 子元素，用于指定当前应用的默认场所；可以包含多个 supported-locale 子元素，用于指定当前应用所支持的各场所。应用的默认场所可以通过调用其 Application 对象的 getDefaultLocale 方法获得；应用所支持的所有场所可调用其 Application 对象的 getSupportedLocales 方法获得。

```
Application app=FacesContext.getCurrentInstance().getApplication();
Locale locale=app.getDefaultLocale();
Iterator<Locale>iter_locale=getSupportedLocales();
```

另外，也需要了解 JSF 框架为视图选择场所的机制，即一个视图应该基于哪种场所来进行呈现。总的来说，JSF 框架会依据用户的请求信息（客户端浏览器能够接受的语言，即请求头 Accept-Language 的值）与应用能支持的场所信息来确定当前视图的场所。具体来说，JSF 框架在创建视图时，将按以下顺序优先为其选择最恰当的场所：

- 上一个视图所用的场所；
- 客户端浏览器能接受的语言与应用支持的场所相匹配的场所；
- 应用的默认场所；
- 支持 JSF 运行的 Java 虚拟机的默认场所。

通过导航请求的新视图总是采用上一个视图的场所，只有在初始请求一个视图时才需要根据请求信息做进一步的选择。客户端浏览器能接受的语言可能有多种，每一种都会与应用支持的各场所进行比较。如果浏览器能接受的所有语言与应用支持的各场所都不能匹配，则选择应用的默认场所。如果应用没有指定默认场所，则选择 Java 虚拟机的默认场所。

JSF 框架选定的场所被保存在当前视图根（UIViewRoot）的 locale 属性中。应用代码也可以根据需要改变该属性。

- public Locale getLocale()；
- public void setLocale(Locale locale)；

当 JSF 框架在计算一些访问资源的 EL 值表达式时，以及标准转换器、验证器在出现处理异常需要错误消息时，都会基于资源包族的基名和该 locale 属性指定的场所、利用格式 2 的 getBundle 方法定位相关的资源包或消息包，进而获取客户端可理解的资源或消息文本。

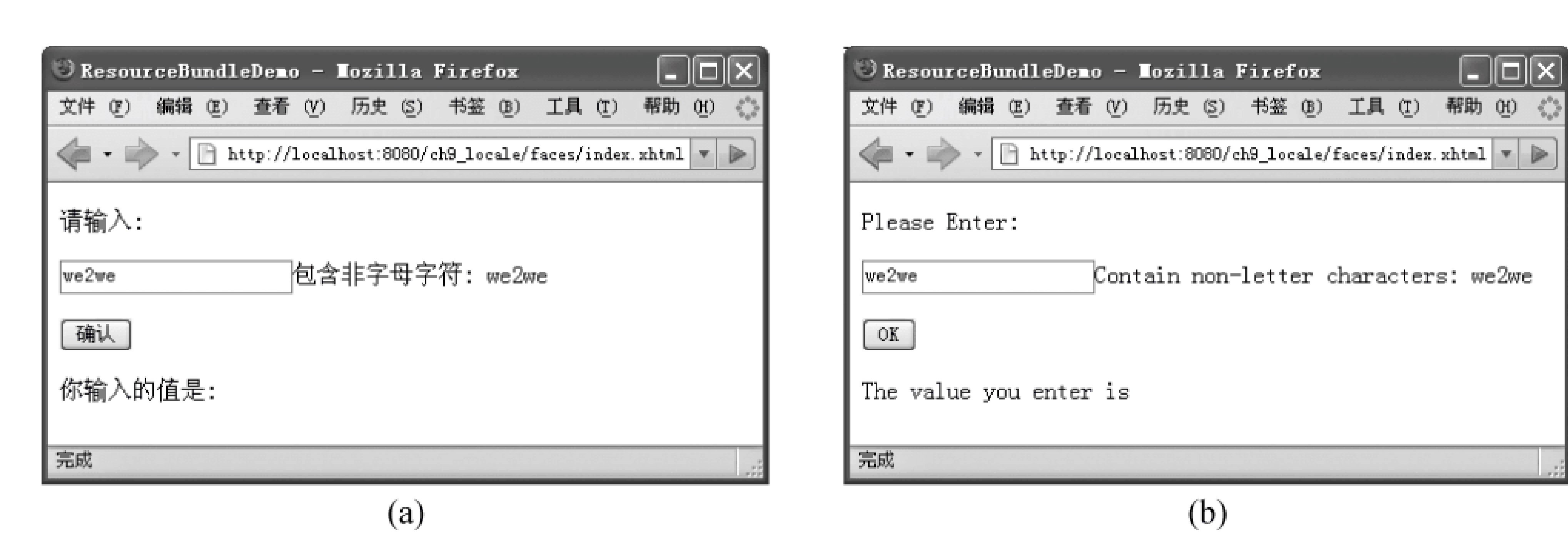
同样的道理，特别是在国际化的 JSF 应用中，自定义转换器、验证器以及事件监听器在出现处理异常需要错误消息时，也应该采用这种方式。否则，如果只是基于资源包族的基名、采用格式 1 的 getBundle 方法定位相关的资源包或消息包，那么只能获取 JSF 应用所在的 Java 虚拟机的默认场所相关的资源和消息。

9.3.5 国际化应用示例

该示例在之前示例（ch9_bundle）的基础上进行修改和扩充而成，用以演示开发基于资

源包的国际化的 JSF 应用的基本过程和方法。为避免影响之前示例的运行效果,这里为该示例新建一个应用项目,命名为 ch9_locale。

该项目仅包含一个 JSF 页面 index. xhtml,其代码与项目 ch9_bundle 中的 index1. xhtml 完全相同。其运行效果如图 9-3 所示。如果客户端浏览器能够接受 zh-cn(中国中文),且优先于 en(英文),那么呈现效果如图 9-3(a)所示,否则呈现效果如图 9-3(b)所示。



(a) (b)

图 9-3 国际化应用 ch9_locale

该应用项目同样包含一个受管 bean 类 bean. Index,与项目 ch9_bundle 中的受管 bean 类 bean. Index 完全相同。

除此之外,作为一个国际化的 JSF 应用,这里主要要为它所支持的场所定义相应的资源包和消息包,并进行必要的声明。另外需要修改自定义验证器,使其能根据与客户端相适应的场所来创建相应的消息对象。

1. 资源包与消息包

首先为应用设置资源包族,并为每个资源包族定义应用所支持场所相应的资源包类或属性文件。该项目包含一个资源包族(BundleOne)和一个消息包族(mymsg),各有两个成员:一个支持 zh_CN,一个支持 en。该资源包族和消息包族的各成员都采用属性文件定义。下面列出各属性文件,其中前面两个是资源包族的成员,后面两个是消息包族的成员。

- bundle/BundleOne_zh_CN. properties。
- bundle/BundleOne_en. properties。
- bundle/mymsg_zh_CN. properties。
- bundle/mymsg_en. properties。

这里,属性文件 BundleOne_zh_CN 包含的资源文本与之前项目(ch9_bundle)中的资源包类 BundleOne. java 定义的资源完全相同,属性文件 BundleOne_en 只是用英文来表达上述资源文本。属性文件 mymsg_zh_CN 的内容与之前项目(ch9_bundle)中的属性文件 mymsg 的内容完全相同,属性文件 mymsg_en 只是用英文来表达这些内容。

有关资源包和消息包的注册声明与之前项目(ch9_bundle)中的注册声明完全相同。

2. 声明应用支持的场所

在 Faces 配置文件中声明应用支持的场所以及应用的默认场所。

```
<application>
    <locale-config>
        <default-locale>en</default-locale>
```

```
        <supported-locale>en</supported-locale>
        <supported-locale>zh_CN</supported-locale>
    </locale-config>
    ......
</application>
```

该应用支持两种场所：en 和 zh_CN。如果客户端浏览器同时支持这两种场所(语言)，就看哪个场所排在请求头 Accept-Language 的值的前面。例如，若请求头 Accept-Language 的值为"en，zh-cn"，那么 JSF 框架为当前视图选择的场所就为 en。如果客户端浏览器既不支持 en，也不支持 zh_CN，那么为当前视图选择的场所将是应用的默认场所，也就是 en。

可以用以下操作设置浏览器可接受的语言：工具|选项(Internet 选项)|语言。

3. 修改自定义验证器

在国际化的 JSF 应用中，自定义转换器、验证器以及事件监听器的编写也需要考虑场所的因素。

该项目包含一个自定义验证器类 validator.MyValidator，它与之前项目(ch9_bundle)中的自定义验证器类基本相同，只是把其中的代码：

```
ResourceBundle mb=ResourceBundle.getBundle(baseName);
```

改为：

```
Locale locale=FacesContext.getCurrentInstance().getViewRoot().getLocale();
ResourceBundle mb=ResourceBundle.getBundle(baseName,locale);
```

之前的代码返回的是与 Java 虚拟机的默认场所相关的资源包，而改进后的代码则返回与客户所在场所相关的资源包。

9.4 小　　结

- 资源包是 ResourceBundle 抽象类的某个具体子类的实例，由一组键—值对组成。
- 资源包不能由 new 表达式创建，而是通过调用 ResourceBundle 类的 getBundle 方法来创建或获取。
- 资源包可以基于 ResourceBundle 抽象类的某个具体子类创建，也可以基于属性文件创建。
- 在 JSF 中，可以在 Faces 配置文件中用 resource-bundle 元素注册资源包，或者在 JSF 页面中用 f:loadBundle 标记装入资源包。
- 一个场所用一个 Locale 类的实例表示，其状态主要由语言和国家(地区)两个数据表示。
- 对一组资源，可以为所支持的各场所创建各自的资源包类(或属性文件)，这些资源包类形成一个资源包族。每个资源包族应该有一个基名。
- 调用 ResourceBundle 类的 getBundle 方法获取的是指定基名和场所的一个资源包链。当应用访问一个资源时，系统会从资源包链中的某个资源包中获取并返回相应的资源。

• 消息包是一种资源包，用于 JSF 的消息机制，可以作为 FacesMessage 消息对象的消息文本来源。一个 JSF 应用只能注册一个消息包(族)。
• 标准转换和验证消息文本都定义在 JSF 框架提供的标准消息包族中，这个标准消息包族的基名是 javax. faces. Messages。

习　题　9

1．简述创建 PropertyResourceBundle 型资源包的方法和过程。

2．简述消息包不同于一般资源包的地方。

3．术语解释：Java 虚拟机默认场所、JSF 应用默认场所。

4．简述 ResourceBundle 类中定义的 getBundle(String)和 getBundle(String，Locale)静态方法的功能区别。

5．创建一个 JSF 应用项目 sh9_locale。项目中包含两个 JSF 页面：index. xhtml 显示登录界面，response. xhtml 是对登录的响应。应用支持中文和英文两种语言，运行效果如图 9-4 所示。

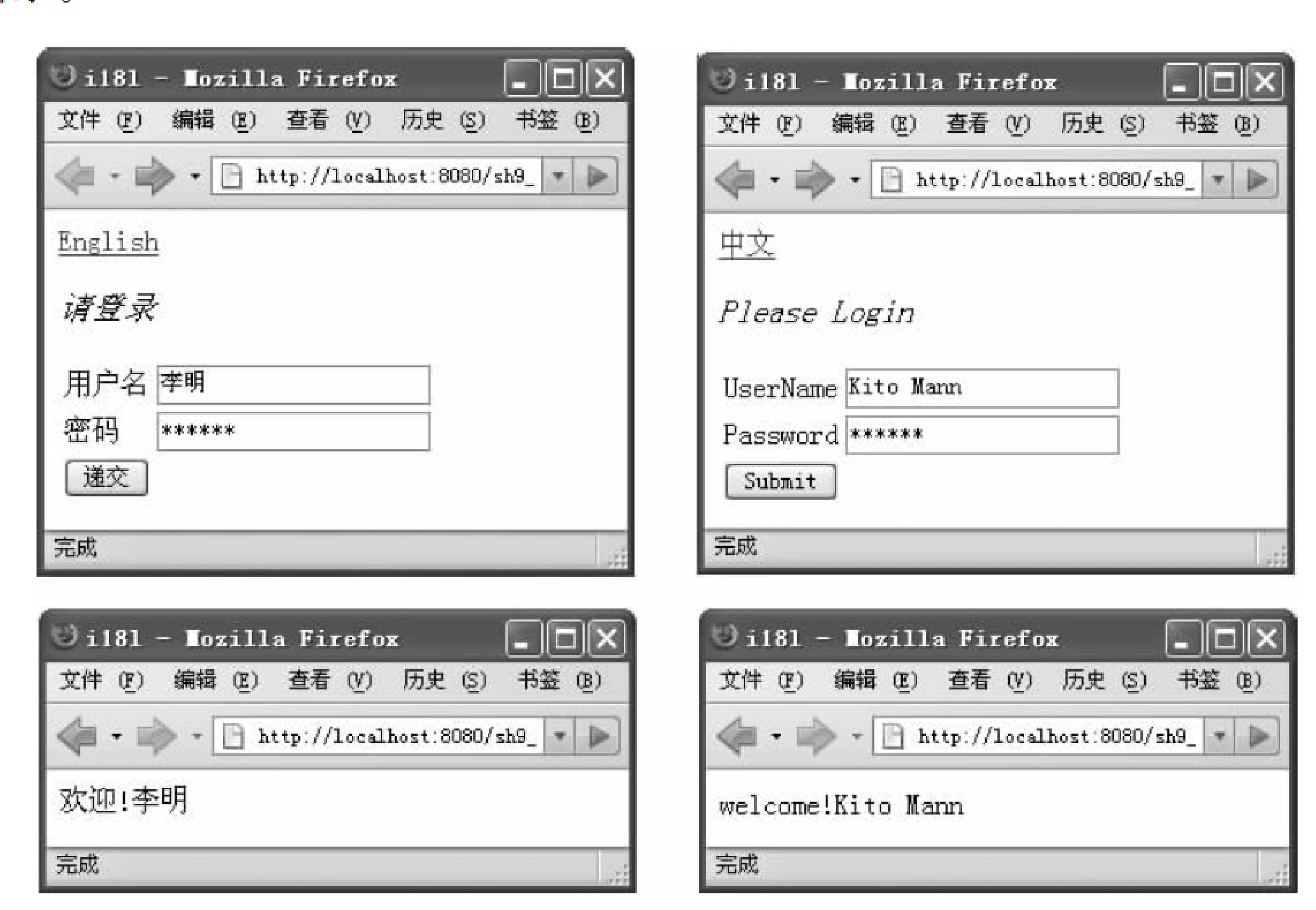


图 9-4　习题 5 示意图

第 10 章　模板与复合组件

本章主题：

- 包含
- 基于模板页创建视图页面
- 基于客户页创建视图页面
- ui:param 与 ui:repeat
- 创建复合组件
- 配置复合组件
- 公开复合组件
- 打包复合组件

相对基于 JSP 视图技术，Facelets 视图技术不仅有一个更好的视图处理器，而且提供模板、复合组件等特性。

模板技术支持页面布局、样式和内容的封装和重用。通过定义和维护封装有页面布局、样式和内容的单个模板，用户可以在多个页面中重用这个模板。这也使得这些页面具有一致的标准外观。

复合组件技术用于将现有的组件和标记组合成一个具有特定功能的可重用的自定义组件。与其他普通的 JSF 组件一样，复合组件也可以进行属性设置，也可以注册所需的验证器、转换器和监听器。

模板和复合组件特性从不同的角度支持对页面内容的封装和重用，对 JSF 应用、尤其是用户界面的开发、设计和维护具有重要的意义。

模板特性主要由 Facelets 标记库支持。要使用 Facelets 标记库，需要在 XHTML 文件中添加一个名称空间声明，如下所示：

```
xmlns:ui="http://java.sun.com/jsf/facelets"
```

该标记库除了支持模板特性，也包含实现其他功能的一些标记。本章 10.1 节至 10.3 节介绍的都是该标记库中的标记。

复合组件特性主要由复合标记库支持。在定义复合组件时，需要在 XHTML 文件中添加如下的名称空间声明：

```
xmlns:cc="http://java.sun.com/jsf/composite"
```

说明：对复合标记库，通常取 composite 作为其空间名称（标记的前缀），但也可以取其他的名称。本书采用 cc 作为复合标记的前缀。

本章 10.4 节至 10.7 节介绍创建和使用复合组件的技术。

10.1 包　　含

在 JSF 中，一个视图页面可以被认为由相对独立的节组成。有些节的内容可能会出现在多个页面中，这时可以将该节内容进行独立封装和定义，称为节内容页。普通的 JSF 页面(可称为包含页)可以将节内容页包含进来，形成完整的视图页面。

通常，节内容页也是一个 XHTML 文件，其中节的内容被包含在 ui：composition 标记内。包含页可以利用 ui：include 标记装入指定的节内容页。

下面通过示例说明包含特性的使用，涉及的文件都存在于 ch10_includeAndTemplate 应用项目。当用户请求包含页(page_one. xhtml)时将呈现如图 10-1 所示的响应。



图 10-1　包含示例运行效果图

代码清单 10-1 给出的是节内容页。这里，真正的节内容就是包含于 ui：composite 标记的文本。

代码清单 10-1　节内容页(section/section_one. xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:head>
    <title>Facelet Title</title>
  </h:head>
  <h:body>
    <ui:composition>
      节内容
    </ui:composition>
  </h:body>
</html>
```

代码清单 10-2 给出的是包含页。其中，ui：include 标记的 src 属性指定节内容页文件。其文件路径应采用相对路径：如果不以斜杠开头，则相对于包含页面文件所在的路径；如果以斜杠(/)开头，则相对于应用的上下文路径(不需要/faces)。

代码清单 10-2　包含页面（page_one.xhtml）

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:head>
    <title>ui:include</title>
  </h:head>
  <h:body>
    <p>包含(ui:include)特性示例</p>
    <ui:include src="section/section_one.xhtml"/>
  </h:body>
</html>
```

当 JSF 框架处理 ui:include 标记时，会把节内容页中 ui:composition 标记包含的内容读取、插入到包含页中 ui:include 标记所在处。节内容页中 ui:composition 标记外面的内容被忽略。

除了重用的目的，对一些较为复杂的 JSF 页面，同样可以采用包含技术，即将一些节内容进行独立封装和定义，而源页面再通过 ui:include 标记将它们包含进来装配成完整的视图页面。这相当于把一个复杂的问题分解成若干简单的小问题来解决，既便于开发、也利于维护。

10.2　Facelets 模板

在 Facelets 模板技术中，一个普通的视图页面被划分为模板页和客户页两部分。模板页定义应用中一些视图页面共有的页面布局、样式和内容，客户页定义某页面视图特有的布局、样式和内容。客户页通过应用模板页形成完整的视图页面。

在模板页中，有些节内容可以被定义成是可变的，可以由客户页提供具体内容。

在 JSF 中，主要有两种使用模板技术的方式，下面分别介绍。

10.2.1　基于模板页创建视图页面

采用这种方式时，一般应首先分析应用中的各视图页面，找出它们共同的布局、样式和内容，定义相应的模板页。然后再为每个视图页面定义相应的客户页，在客户页中调用模板页，并定义其特有的布局、样式和内容。

在这种方式中，模板页是形成最终视图页面的基础，应包含一个视图页面应该有的各部分内容。

下面通过示例说明如何采用该种方式使用 Facelets 模板特性，其中涉及的文件都存在于 ch10_includeAndTemplate 应用项目。当用户请求客户页（client_one.xhtml）时将呈现图 10-2 所示的响应。

图 10-2　模板应用示例运行效果图

代码清单 10-3 给出的是模板页。其中共含 3 个 ui:insert 标记，每个标记定义了一个内容可变的节，每个节的内容可由客户页提供。以第一个 ui:insert 标记为例，该标记定义了一个名称为 top 的节(由标记的 name 属性指定)，节中有默认的内容，即 DefaultTop。如果客户页没有为该节提供具体的内容，那么最终的视图页面将默认该内容，否则显示客户页提供的具体内容。

代码清单 10-3　模板页(template/template_one.xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <h:outputStylesheet library="css" name="layout.css"/>
    <h:outputStylesheet library="css" name="default.css"/>
    <title>Facelets Template_1 </title>
  </h:head>
  <h:body>
    <div id="top">
      <ui:insert name="top">DefaultTop</ui:insert>
    </div>
    <div id="content">
      <ui:insert name="content">DefaultContent</ui:insert>
    </div>
    <div id="bottom">
      <ui:insert name="bottom">DefaultBottom</ui:insert>
    </div>
  </h:body>
</html>
```

该模板页使用了两个层叠样式表，即 layout. css 和 default. css。这两个样式表都存放在应用文档根目录下的 resources/css 子目录下。代码清单 10-4 和代码清单 10-5 列出它们定义的样式。

代码清单 10-4　用于布局的样式（layout. css）

```
#top {
  position: relative;
  background-color: #036fab;
  color: white;
  padding: 5px;
  margin: 0px 0px 10px 0px;
}
#bottom {
  position: relative;
  background-color: #c2dfef;
  padding: 5px;
  margin: 10px 0px 0px 0px;
}
#content {
  position: relative;
  background-color: #dddddd;
  padding: 5px;
}
```

代码清单 10-5　默认的样式（default. css）

```
body {
  background-color: #ffffff;
  font-size: 14px;
  color: #000000;
  margin: 10px;
}
```

接下来介绍客户页。在这种方式中，客户页主要为模板页中由 ui:insert 标记定义的节提供具体的内容。代码清单 10-6 给出了该示例的客户页。

代码清单 10-6　客户页（client_one. xhtml）

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html">
  <body>
    <ui:composition template="template/template_one.xhtml">
      <ui:define name="top">
        <h:outputStylesheet library="css" name="css_one.css"/>
```

```
        <div class="top">
          页头
        </div>
      </ui:define>
      <ui:define name="content">
        <ui:include src="section\login.xhtml"/>
      </ui:define>
      <ui:define name="bottom">
        <div class="bottom">
          页脚
        </div>
      </ui:define>
    </ui:composition>
  </body>
</html>
```

在客户页中，为模板页中的节提供的具体内容应包含在 ui:define 标记内。ui:define 标记的 name 属性指定为哪个节提供内容，即该属性值应与模板页中的某个 ui:insert 标记的 name 属性值相匹配。

在客户页中，所有的 ui:define 标记都必须嵌套于带 template 属性的 ui:composition 标记。当 JSF 框架处理客户页的 ui:composition 标记时，首先会丢弃标记外的内容，然后装入 template 属性指定的模板页，最后用客户页中各 ui:define 标记内的内容替换模板页中对应的 ui:insert 标记、形成最终的视图页面。

该示例中，客户页为名称为 content 的节提供的具体内容实际定义在一个节内容页中，客户页通过 ui:include 标记把它包含进来。该节内容页的代码见代码清单 10-7。

代码清单 10-7　节内容页（section/login.xhtml）

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:head>
    <title>login</title>
  </h:head>
  <h:body>
    <ui:composition>
    <h:form>
    <h:outputLabel for="name" value="用户名"/>
    <h:inputText id="name"/>
    <p/>
    <h:outputLabel for="pw" value="密 码"/>
    <h:inputSecret id="pw"/>
    <p/>
```

```
    <h:commandButton value="OK"/>
    </h:form>
    </ui:composition>
  </h:body>
</html>
```

ui:include 标记不仅可以用于普通的 JSF 页(包含页)、客户页,也可用于模板页,比如可以在模板页的 ui:insert 标记内用 ui:include 标记装入该节的默认内容。

该示例中,客户页也使用了一个层叠样式表,即 css_one.css(代码清单 10-8)。该样式表也存放在应用文档根目录下的 resources/css 子目录下,其定义的样式主要用于页头和页脚的显示,可以覆盖模板中已定义的默认样式。

代码清单 10-8　客户页定义的样式(css_one.css)

```
.top {
  text-align: center;
  font-size: 36px
}
.bottom {
  text-align: center;
  font-size: 12px
}
```

说明:在该示例的客户页中,把链接层叠样式表的<h:outputStylesheet>标记放置在某个<ui:define>标记内的原因是:模板页中只定义了有名的节,这样 JSF 在处理客户页时,就只处理<ui:define>标记内的内容,而不会处理<ui:composition>标记内、<ui:define>标记外的内容。

模板页、客户页和节内容页都是 xhtml 文件,可以像普通 JSF 页一样来创建,并存放在应用的文档根目录或某个子目录下。比如为便于管理,将模板页存放在 template 子目录下,将节内容页存放在 section 子目录下等。

对基于模板页创建视图页面的方式,NetBeans IDE 提供了专门的命令,可以帮助用户更加方便地创建模板页和客户页。

- 创建一个指定布局样式的模板页:
  新建文件|JavaServer Faces|Facelets 模板。
- 创建一个使用指定模板页的客户页:
  新建文件|JavaServer Faces|Facelets 模板客户端。

10.2.2　基于客户页创建视图页面

一般来说,这种方式适用于在已有的页面中引入一些公共的布局、样式和内容。这些共同的布局、样式和内容定义于模板页,而原有的页面通过 ui:decorate 标记引用模板页,被称为客户页。

在这种方式中,客户页是形成最终视图页面的基础,应包含一个视图页面应该有的各部分内容。模板页可用于装饰客户页中指定的内容。

下面通过示例说明如何采用该种方式使用 Facelets 模板特性，其中涉及的文件都存在于 ch10_includeAndTemplate 应用项目。有些文件与 10.2.1 节所举示例的文件完全相同，下面不再列出。当用户请求客户页(client_two.xhtml)时呈现的响应也与 10.2.1 节所举示例的呈现响应相同，如图 10-2 所示。

先来看客户页(代码清单 10-9)。可以设想客户页本来仅包含一个表单，现在需要为其添加一个页头和页脚，并可对表单的显示引入某些额外的样式。为此，可以定义一个模板页(template_two.xhtml)，模板页中包含有页头、页脚以及相关的样式表，然后在客户页中用 ui.decorate 标记引用模板页，让模板页来装饰客户页原有的内容，即表单。

代码清单 10-9　客户页(client_two.xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:head>
    <h:outputStylesheet library="css" name="default.css"/>
    <title>Facelets Template_2</title>
  </h:head>
  <h:body>
    <ui:decorate template="template\template_two.xhtml">
    <h:form>
    <h:outputLabel for="name" value="用户名"/>
    <h:inputText id="name"/>
    <p/>
    <h:outputLabel for="pw" value="密　码"/>
    <h:inputSecret id="pw"/>
    <p/>
    <h:commandButton value="OK"/>
    </h:form>
    </ui:decorate>
  </h:body>
</html>
```

在这种方式中，虽然形成最终视图页面的基础是客户页，但形成客户页中 ui:decorate 标记最终呈现内容的基础是模板页。接下来看模板页(代码清单 10-10)。在这里，模板内容实际定义于模板页的 ui:composition 标记内，其中包含三个节(由 ui:insert 标记定义)，第 1 个节和第 3 个节的名称分别为 top 和 bottom，并都提供了默认内容，第 2 个节没有名称，也没有定义默认内容。

代码清单 10-10　模板页(templat_two.xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ui="http://java.sun.com/jsf/facelets">
  <h:head>
    <title>Facelet Title</title>
  </h:head>
  <h:body>
    <ui:composition>
      <h:outputStylesheet library="css" name="layout.css"/>
      <h:outputStylesheet library="css" name="css_one.css"/>
      <ui:insert name="top">
        <div id="top" class="top">
        页头
        </div>
      </ui:insert>
      <div id="content">
        <ui:insert/>
      </div>
      <ui:insert name="bottom">
        <div id="bottom" class="bottom">
        页脚
        </div>
      </ui:insert>
    </ui:composition>
  </h:body>
</html>
```

当 JSF 框架处理客户页的 ui:decorate 标记时，将读取指定模板页中 ui:composition 标记内的内容(标记外的内容被丢弃)，并以此为基础形成客户页 ui:decorate 标记最终要呈现的内容。如果客户页 ui:decorate 标记内有 ui:define 标记，则其内容将代替模板页中相应的 ui:insert 标记定义的节的默认内容。本示例不存在这样的 ui:define 标记，所以模板页中定义的 top 节和 bottom 节都将呈现其默认内容。

在该示例中，模板页中无名节的定义，即＜ui:insert/＞，显得尤为重要。如果没有定义该无名节，那么客户页中 ui:decorate 标记中原先的内容(即表单)将不会出现在最终视图中。而一旦定义了无名节，那么客户页 ui:decorate 标记内、ui:define 标记外的所有内容就会作为模板页中定义的无名节要呈现的内容。

10.3 ui:param 与 ui:repeat

这里介绍 Facelets 标记库的另外两个标记：ui:param 与 ui:repeat。

10.3.1 ui:param 标记

在包含特性和模板特性中，节内容页和模板页作为可重用的语法成分，通常会在多处被

调用。为增强其通用性，Facelets 允许包含页向节内容页、客户页向模板页传递参数。

包含页和客户页通过 ui:param 标记向节内容页和模板页传递参数，此时该标记应该作为 ui:include 标记、带 template 属性的 ui:composition 和 ui:decorate 标记的子标记。

ui:param 标记有两个属性，name 属性指定参数名，value 属性指定参数值。比如下面是包含页的一段代码：

```
<ui:include src=……>
    <ui:param name="currentDate" value="#{myBean.currentDate}"/>
<ui:include/>
```

被传递的参数作为一个 EL 变量，可以用于节内容页、模板页中的 EL 表达式中。比如，下面是节内容页的一段代码：

```
<div class="c1">
    Today's date: #{currentDate}
</div>
```

10.3.2 ui:repeat 标记

与 h:dataTable 标记相似，ui:repeat 标记也能对数组或表中的元素进行迭代处理。相比两者，h:dataTable 标记适合将数组或表中的数据呈现成数据表格，而 ui:repeat 标记则可以更自由或自主地去呈现数组或表中的数据。

下面代码演示了 ui:repeat 标记的用法。其中，value 属性通过 EL 表达式指向该标记要迭代处理的数组或表，这里假设 books 是受管 bean 中类型为 List<Book>的属性。var 属性暴露一个变量，表示当前被迭代处理的元素。

```
<ui:repeat value="#{myBean.books}" var="book">
    <p>
    <h:outputText value="书　名"/>
    <h:outputText value="#{book.title}"/><br/>
    <h:outputText value="作　者"/>
    <h:outputText value="#{book.author}"/><br/>
    <h:outputText value="出版社"/>
    <h:outputText value="#{book.publisher}"/>
    </p>
</ui:repeat>
```

除了 value 和 var 属性，ui:repeat 标记还包含以下属性，允许迭代处理数组或表的子集。

- offset：迭代处理的首元素的索引，默认值为 0。
- step：步长。默认值为 1，即连续处理各元素。
- size：迭代处理的尾元素的索引，默认值为数组或表中最后一个元素的索引。

例如，books 表的大小为 10(10 个元素的索引依次为 0，1，2，…，9)，那么下面标记会迭代处理的元素的索引应包括：3、5、7。

```
<ui:repeat value="#{myBean.books}" var="book" offset="3" step="2" size="8">
```

另外，ui:repeat 标记的 varStatus 属性暴露一个变量，该变量指向一个对象，对象包含一组只读的表示迭代状态的 JavaBean 属性。表 10-1 列出了这些 JavaBean 属性。

表 10-1　表示迭代状态的 JavaBean 属性

JavaBean 属性	含　义	类　型
index	当前元素在数组或表中的索引	int
begin	对应标记的 offset	Integer
end	对应标记的 size	Integer
step	对应标记的 step	Integer
even	当前元素是否为第偶数个被迭代的元素	boolean
odd	当前元素是否为第奇数个被迭代的元素	boolean
first	当前元素是否是第一个被迭代的元素	boolean
last	当前元素是否是最后一个被迭代的元素	boolean

例如，下面标记在呈现每本图书时，会先显示该图书在表或数组中的索引。

```
<ui:repeat value="#{myBean.books}" var="book" varStatus="status">
    #{status.index}<br/>
    ……
</ui:repeat>
```

10.4　创建复合组件

与普通的 JSF 页一样，复合组件也是一个 XHTML 文件，称为复合组件页。每个复合组件页定义一个复合组件。复合组件页应放置在应用文档根目录下的 resources 子目录下的某个子目录中，如 ezcomp，这个子目录被看作是一个复合组件库。

在 NetBeans IDE 中，可以调用以下命令创建一个复合组件：

```
新建文件|JavaServer Faces|JSF 复合组件
```

复合组件的定义包括接口和实现两部分。实现定义于 cc:implementation 标记内，通常是根据需要由一些标准的或现有的 JSF 标记组合而成。接口定义于 cc:interface 标记内，用于向用户公开该复合组件的用法、可配置属性等。

下面通过一个有关登录复合组件的示例说明复合组件的定义、使用等技术。该示例的完整代码保存在应用项目 ch10_compositedemo 中。

代码清单 10-11 列出登录复合组件的定义，其实现定义主要由组成登录表单的一些标准的 JSF 标记组合而成。该文件保存在 resources 目录下的 ezcomp 子目录中。

代码清单 10-11　登录复合组件(login.xhtml)基本版

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:cc="http://java.sun.com/jsf/composite"
    xmlns:h="http://java.sun.com/jsf/html">
  <cc:interface/>
  <cc:implementation>
    <p>
    请登录
    </p>
    <h:form id="form">
    <h:panelGrid columns="2">
      用户名
      <h:panelGroup>
        <h:inputText id="name" value="#{myBean.name}" required="true"/>
        <h:message for="name"/>
      </h:panelGroup>
      密码
      <h:panelGroup>
        <h:inputSecret id="pw" value="#{myBean.password}" required="true"/>
        <h:message for="pw"/>
      </h:panelGroup>
    </h:panelGrid>
    <p>
      <h:commandButton id="button" value="登录" action="#{myBean.action}"/>
    </p>
    </h:form>
  </cc:implementation>
</html>
```

这是登录复合组件定义的基本版，没有提供任何可配置的属性。这里主要是为了说明复合组件的定义过程、组成和使用等，关于复合组件定义的更多内容会在后面陆续介绍。

说明：在上述定义中，<cc:interface>标记虽然没有指定任何属性，也没有包含任何子标记，但该标记本身不能缺省。

定义好了复合组件，就可以在普通的 JSF 页面中使用复合组件。要使用复合组件，需要在 JSF 页中声明复合组件库的名称空间，例如：

```
xmlns:ezcomp="http://java.sun.com/jsf/composite/emcomp"
```

名称空间的值必须以 http://java.sun.com/jsf/composite/开头，其余的内容必须与复合组件库对应的子目录名一致。可以为名称空间指定任意名称，但通常取复合组件库对应的子目录名，如 ezcomp。

代码清单 10-12 演示了如何使用登录复合组件。

代码清单 10-12　使用登录复合组件(index.xhtml)

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:ezcomp="http://java.sun.com/jsf/composite/ezcomp">
 <h:head>
   <title>composite demo</title>
 </h:head>
 <h:body>
   <ezcomp:login/>
 </h:body>
</html>
```

10.5 配置复合组件

复合组件作为一种可重用的软件成分应该是可配置的，以便能在不同的场合被方便而又恰当地使用。上述登录复合组件的定义显然并不理想，因为所有内容都是固定而不可配置的。例如，要使用这个复合组件，必须存在相应的名为 myBean 的受管 bean，其中定义了可读写的 name 属性和 password 属性，以及一个名为 action 的动作方法。

可以在复合组件的接口定义中为复合组件声明相应的属性，然后在复合组件的实现定义中利用这些属性来编写复合组件的相关内容，而在使用复合组件时再为这些属性指定具体的值。显而易见，这样的复合组件是可配置的，具有更好的可重用性。

用 cc:attribute 标记声明复合组件的属性，该标记只能用于 cc:interface 标记内。也就是说，应该在复合组件的接口定义中声明复合组件的属性。下面是 cc:attribute 标记的一些常用属性。

- name：为定义属性指定属性名。
- required：指明在使用复合组件时是否必须为定义属性指定值。
- default：为定义属性指定一个默认值。
- type：指定定义属性必须指向一个值表达式，该属性指定值表达式的类型。
- method-signature：指定定义属性必须指向一个方法表达式，该属性指定方法表达式的签名。这里，方法签名应包括方法返回类型、方法名和形参。返回类型和形参类型可以是基本类型也可以是引用类型，如果是引用类型，应给出完整类名。方法名可以是一个任意的别名，在使用复合组件、设置属性时，再指定具体的方法名。

如果 type 和 method-signature 属性都没有指定，则一般认为定义属性值是一个值表达式，其类型为 java.lang.Object。如果两个属性都指定了，则 method-signature 属性被忽略。

一旦声明了属性，在复合组件的实现定义中就可以通过 EL 表达式引用这些属性。引用属性的一般格式为：#{cc.attrs.＜属性名＞}。其中 cc 表示当前复合组件，aatrs 表示该复合组件的属性集。

代码清单 10-13 给出了可配置的登录复合组件的定义，文件名为 login_1.xhtml。除用户名文本域组件的值、口令域组件的值和动作按钮的动作方法，表单标题、文本域标签、口令域标签和按钮标题都是可配置的。

代码清单 10-13　可配置的登录复合组件定义(login_1.xhtml)

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:cc="http://java.sun.com/jsf/composite"
    xmlns:h="http://java.sun.com/jsf/html">
  <cc:interface>
    <cc:attribute name="formTitle" default="Please Log in"/>
    <cc:attribute name="namePrompt" default="Username"/>
    <cc:attribute name="pwPrompt" default="Password"/>
    <cc:attribute name="buttonTitle" default="Login"/>
    <cc:attribute name="name" required="true"/>
    <cc:attribute name="password" required="true"/>
    <cc:attribute name="loginAction" method-signature="java.lang.String m()"
        required="true"/>
  </cc:interface>
  <cc:implementation>
    <p>
    #{cc.attrs.formTitle}
    </p>
    <h:form id="form">
      <h:panelGrid columns="2">
        #{cc.attrs.namePrompt}
        <h:panelGroup>
          <h:inputText id="name" value="#{cc.attrs.name}" required="true"/>
          <h:message for="name"/>
        </h:panelGroup>
        #{cc.attrs.pwPrompt}
        <h:panelGroup>
          <h:inputSecret id="pw" value="#{cc.attrs.password}" required="true"/>
          <h:message for="pw"/>
        </h:panelGroup>
      </h:panelGrid>
      <p>
      <h:commandButton id="button" value="#{cc.attrs.buttonTitle}"
            action="#{cc.attrs.loginAction}"/>
      </p>
    </h:form>
  </cc:implementation>
</html>
```

现在,页面制作者可以更加灵活地使用该登录复合组件了。其中,name 属性、password 属性和 loginAction 属性是必需的,使用者可以根据具体的情况为它们指定合适的值表达式和方法表达式。其他属性是可选的,但在声明时都为它们指定了各自的默认值。下面代码是使用该登录复合组件的一个示例。

```
<ezcomp:login_1 name="#{myBean.name}" password="#{myBean.password}"
                    loginAction="#{myBean.action}"/>
```

10.6 公开复合组件

可以为复合组件注册所需的转换器、验证器或事件监听器，但这些转换器、验证器等总是针对复合组件中的某个组件的，而不大可能是为整个复合组件服务的。例如，可以为登录复合组件注册一个验证器用以验证其中的密码是否正确，可以为该复合组件注册一个动作事件监听器用以监听登录按钮引发的动作事件等。

一种方法是在复合组件的实现定义中，为复合组件中的某个组件注册适当的转换器、验证器或事件监听器，但这种方式较为死板，是不可配置的。较为灵活的方式应该是在使用复合组件时，决定是否为某组件注册转换器、验证器或事件监听器，以及注册何种转换器、验证器和事件监听器。要做到这一点，必须先公开复合组件中的相关组件。通过公开复合组件中的组件，页面制作者就可以在使用复合组件时针对其中的某个具体组件注册相关的转换器、验证器或事件监听器。

公开复合组件属于使用复合组件的约定的范畴，要在复合组件的接口定义中进行。下面标记用于公开复合组件中某种类型的组件。

- cc:editableValueHolder：公开 EditableValueHolder 型组件，如输入类组件。
- cc:valueHolder：公开 ValueHolder 型组件，如输入类组件、输出类组件。
- cc:actionSource：公开 ActionSource 型组件，如动作按钮、动作超链接等。

上述标记都有两个属性：name 和 targets。其中 targets 属性指定要公开的组件的客户端标识符。可以指定多个组件的客户端标识符，此时各标识符之间用空格分隔。name 属性为公开的一个或多个组件指定一个名称。使用复合组件时，可以通过该名称为相应的组件注册所需的转换器等。

代码清单 10-14 给出了新的登录复合组件的定义，其文件名为 login_2.xhtml。与上一个版本相比，它仅在接口定义中增加了三行，用于公开相关组件。

代码清单 10-14 新版登录复合组件定义(login_2.xhtml)

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:cc="http://java.sun.com/jsf/composite"
    xmlns:h="http://java.sun.com/jsf/html">
  <cc:interface>
    ......
    <cc:editableValueHolder name="nameInput" targets="form:name"/>
    <cc:editableValueHolder name="passwordInput" targets="form:pw"/>
    <cc:editableValueHolder name="inputs" targets="form:name form:pw"/>
  </cc:interface>
  <cc:implementation>
    ......
  </cc:implementation>
</html>
```

这里，nameInput 是用户名文本域的公开名称，passwordInput 是口令域的公开名称，而 inputs 则同时表示上述两个组件。

现在，页面制作者可以在使用登录复合组件时，为这些输入类组件注册所需的转换器、验证器了。下面代码是使用该登录复合组件的一个示例，其中为用户名文本域和口令域注册了相关的验证器。注册标记中的 for 属性指定目标组件的公开名称，即要与公开复合组件标记中的 name 属性值相匹配。

```
<ezcomp:login_2 name="#{myBean.name}"
          password="#{myBean.password}"
          loginAction="#{myBean.action}">
   <f:validateLength maximum="15" for="inputs"/>
   <f:validateLength minimum="4" for="nameInput"/>
   <f:validator validatorId="pwValidator" for="passwordInput"/>
</ezcomp:login_2>
```

注册的两个长度验证器规定：用户名和密码最多输入 15 个字符；用户名至少输入 4 个字符。注册的第三个验证器是一个自定义验证器，用于验证密码值是否正确。代码清单 10-15 列出了该自定义验证器的代码，其功能是确保密码只能由数字和字母组成。

代码清单 10-15　验证器(PasswordValidator.java)

```
package validator;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.FacesValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

@FacesValidator("pwValidator")
public class PasswordValidator implements Validator {
  public void validate(FacesContext context,UIComponent component,Object value)
                        throws ValidatorException {
    StringBuilder sb=new StringBuilder((String)value);
    int i=0;
    while(i<sb.length()){
      char c=sb.charAt(i);
      if(!(Character.isDigit(c)||Character.isLetter(c))){
        String s="密码中包含非数字、字母";
        FacesMessage msg=new FacesMessage(s,s);
        throw new ValidatorException(msg);
      }
      i++;
    }
  }
}
```

说明：除了注册验证器标记(f:validator 等)，注册转换器标记(f:converter 等)和注册动作监听器标记(f:actionListener)也都有一个 for 属性。如果使用 for 属性，那么这些注册

标记应该出现在复合组件标记内，for属性指定复合组件中的某个公开的组件。

10.7 将复合组件打包成JAR文件

可以将复合组件打包成JAR文件，以便在其他JSF应用中使用这些复合组件。复合组件的相关文件通常包括：

- 存放在复合组件库目录（如ezcomp）下的复合组件定义文件（如login_1.xhtml）；
- 复合组件定义用到的层叠样式表文件、图像文件等。

这些文件原本都应存放在resources目录下。打包复合组件时，resources目录及其下面的与复合组件相关的文件都应按原先的位置放置在JAR文件中的META-INF子目录下，如图10-3所示。

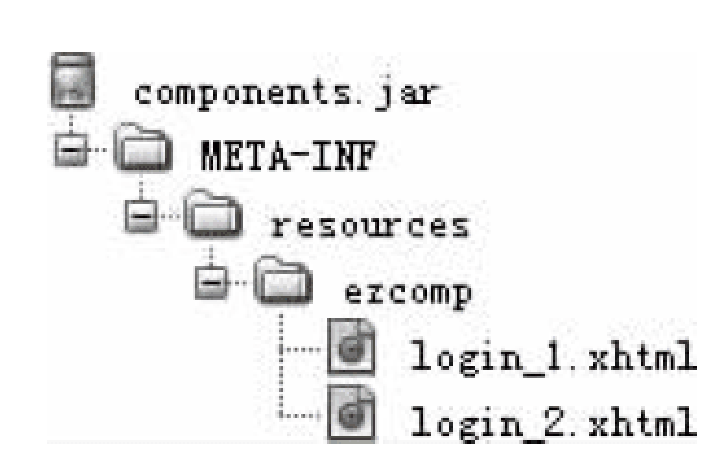


图10-3 打包复合组件文件

按以下步骤可以将复合组件打包成JAR文件：

（1）创建一个空的工作目录，然后在其中创建META-INF子目录；

（2）将resources目录、复合组件库以及相关资源（样式表等）按它们原来的位置复制到METE-INF子目录；

（3）打开DOS命令窗口，将上述工作目录设置为当前目录，然后输入以下jar命令产生JAR文件：jar -cvfM components.jar *.*。

10.8 小　　结

- 包含页可以利用ui:include标记装入指定的节内容页。节内容页也是一个XHTML文件，其中节的内容被包含在ui:composition标记内。
- 使用Facelets模板技术的方式有两种：基于模板页创建视图页面和基于客户页创建视图页面。
- 在基于模板页创建视图页面方式中，模板页是形成最终视图页面的基础，应包含一个视图页面应该有的各部分内容，只是有些节的内容可由客户页提供。
- 在基于客户页创建视图页面方式中，客户页是形成最终视图页面的基础，应包含一个视图页面应该有的各部分内容，模板页可用于装饰客户页中指定的内容。
- 与h:dataTable标记相似，ui:repeat标记也能对数组或表中的元素进行迭代处理。相比h:dataTable标记，ui:repeat标记可以更自由或自主地去呈现数组或表中的数据。
- 复合组件也是一个XHTML文件，称为复合组件页。每个复合组件页定义一个复合组件，包括接口定义和实现定义两部分。
- 实现定义于cc:implementation标记内，通常是根据需要由一些标准的、或现有的JSF标记组合而成。接口定义于cc:interface标记内，用于向用户公开该复合组件的用法、可配置属性等。
- 复合组件能够被定义成可配置的，使得使用者可以为复合组件指定相关的属性，或

者为复合组件的某个组成组件指定所需的转换器、验证器或事件监听器等。

习 题 10

1. 比较 ui:include 标记、带 template 属性的 ui:composition 和 ui:decorate 标记的功能。

2. 比较 ui:param 标记与 f:param 标记的功能。

3. 简述创建复合组件的基本过程和方法。

4. 何谓公开复合组件？举例说明公开复合组件及使用公开的信息的方法。

5. 利用 Facelets 模板技术为应用项目 sh6_lookandbuy(第 6 章习题 6)中的每个页面增加所需的页头和页脚。

第 11 章　Java DB 与实体类

本章主题

- Java DB 基本操作
- 基本的 SQL 语句
- JPA 概述
- 实体类
- 通过数据库生成实体类

大多数 Web 应用都离不开数据的存储和访问。在 Java EE 中,JPA 是数据持久层的标准。利用 JPA,一个 Web 应用可以实现数据的存储、访问、更新和删除。

JPA 技术建立在 JDBC 技术之上。对不同的数据库系统需要配置不同的 JDBC 驱动程序,但对于应用程序来说,则可以采用统一的编程接口。也就是说,相同的程序代码在不同的 JDBC 驱动程序支持下,可以访问不同的数据库系统。

JPA 的内容主要涉及实体类的定义、实体管理器 API 和 JPQL 等三方面。本章首先简单介绍 JDK 自含的数据库管理系统 Java DB,然后介绍实体类的定义及相关操作。有关实体管理器 API 和 JPQL 的内容将在下一章介绍。

11.1　Java DB

Java DB 源于 Apache 软件基金会(Apache Software Foundation, ASF)名下的 Derby 项目,是一个纯 Java 实现、开源的数据库管理系统。Java DB 小巧但功能完备,支持外键、索引、视图、触发器、存储过程、事务处理等大部分数据库应用所需的特性。

Java DB 是 Java SE 6 平台的组成部分,Java 程序员不再需要耗费大量精力安装和配置数据库管理系统,就能进行安全、易用、标准、并且免费的数据库编程。NetBeans IDE 及 Glassfish 同样提供了对 Java DB 的支持。

11.1.1　基本操作

在 NetBeans IDE 的"服务"窗口中,用户可以方便地进行 Java DB 数据库的创建、连接和执行 SQL 命令等操作。

1. 创建数据库

在"服务"窗口中,有一个"数据库"节点,其中包含"Java DB"、"驱动程序"等子节点。右击"Java DB"节点,选择打开的快捷菜单项,可以启动或停止 Java DB 数据库服务器、创建 Java DB 数据库以及设置存放数据库的位置等。

设置存放数据库的位置的操作步骤如下。

(1) 右击"Java DB"节点,选择"属性"命令,打开"Java DB 属性"对话框。

(2) 在"数据库位置"文本域中输入用于存放数据库的路径,或者单击右侧的"浏览"按

钮选择相应的路径。

(3) 单击“确定”按钮。

创建一个 Java DB 数据库的操作步骤如下:

(1) 右击“Java DB”节点,选择“创建数据库”命令,将打开“创建 Java DB 数据库”对话框,如图 11-1 所示。

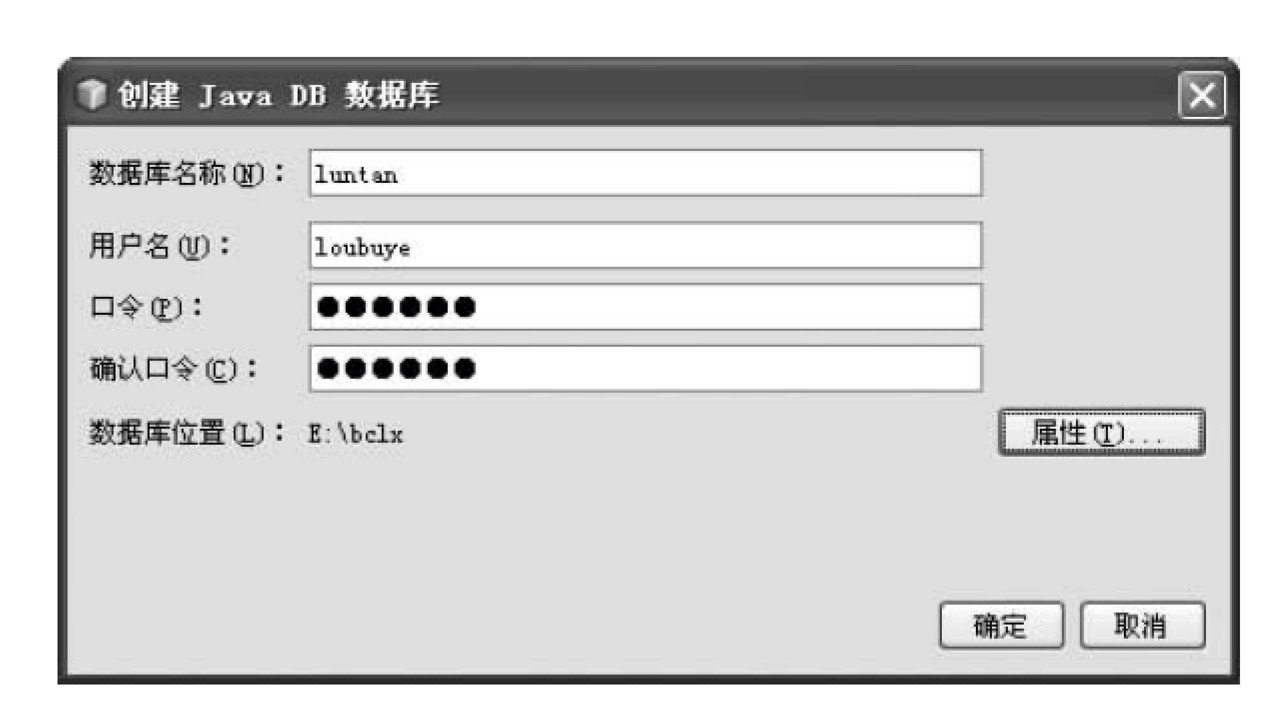


图 11-1 创建 Java DB 数据库

(2) 在对话框中设置数据库名称以及用户名和口令。在这里,单击“属性”按钮,同样会打开“Java DB 属性”对话框,也可以设置存放数据库的位置。

(3) 单击“确定”按钮。

新建一个 Java DB 数据库后,NetBeans IDE 会自动创建一个该数据库的连接节点,数据库连接节点是“数据库”节点的子节点。每个连接节点包含了与特定数据库连接所需要的相关信息,如驱动程序名称、数据库服务器所处的主机域名、端口号等。

2. 连接数据库与执行 SQL 语句

NetBeans IDE 为新建数据库自动创建连接节点并不意味着与该数据库建立了连接。要对数据库进行操作,首先应该与数据库建立连接。

建立从 IDE 至数据库的连接的方法是:右击相应的数据库连接节点,然后在打开的快捷菜单中选择“连接”命令。若连接成功,该连接节点下会出现数据库的架构(又称模式)节点,每个架构节点下会包含“表”、“视图”和“过程”3 个子节点。

建立数据库连接后,可以打开“SQL 命令”窗口,实现对数据库的操作。使用“SQL 命令”窗口的方法如下:

(1) 右击一个数据库连接节点或该节点下的“表”子节点,选择“执行命令”选项,将打开“SQL 命令”窗口,如图 11-2 所示。该窗口一般显示于编辑器窗格内。

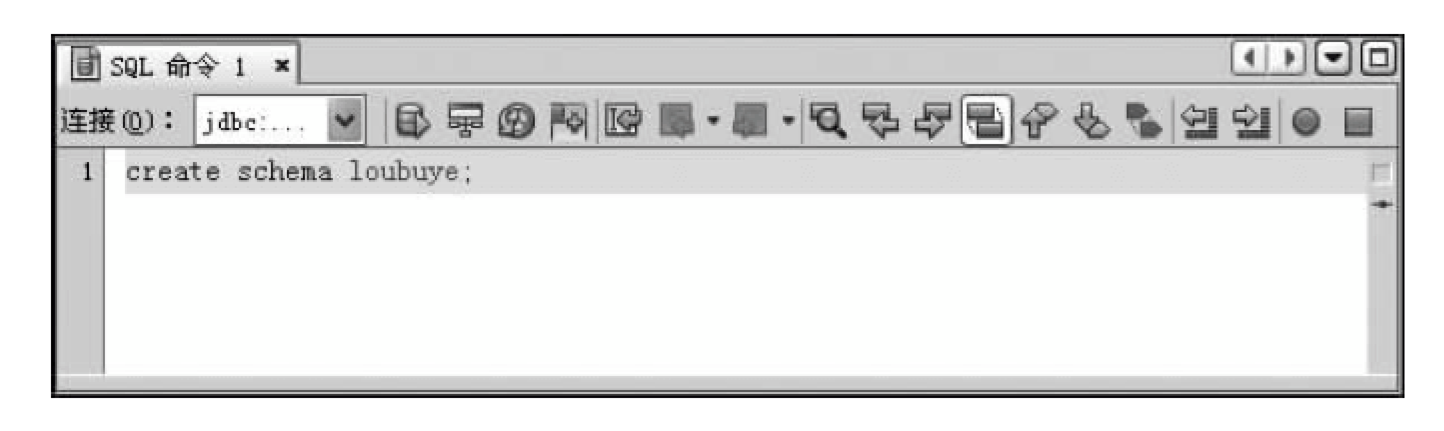


图 11-2 SQL 命令窗口

在“SQL 命令”窗口中,“连接”下拉框内应显示要操作的目标数据库的对应连接。

(2) 在“SQL 命令”窗口中输入一条或多条 SQL 语句(各语句之间用分号隔开)。

（3）单击“运行 SQL”按钮，执行窗口中的 SQL 语句。

SQL 语句执行后的结果信息会显示在“输出”窗口。如果执行的是 SELECT 语句，查询结果将显示于“SQL 命令”窗口的下方。

NetBeans IDE 自动创建的数据库连接节点，设定与用户名同名的架构为数据库的缺省架构。如果该缺省架构不存在，可以执行下面 SQL 语句创建之，其中 schema-name 取用户名。

```
CREATE SCHEMA schema-name
```

虽然右击某架构节点，然后选择“设置为缺省架构”命令可以改变数据库的缺省架构，但数据库连接节点中设定的缺省架构是不会改变的。一些自动工具在访问数据库时，仅会列出连接节点中设定的缺省架构中的表，所以一般应确保连接节点中设定的缺省架构在数据库中是存在的。

3. 创建数据库连接节点

右击数据库连接节点，选择“连接”命令，可以建立与数据库的连接。建立连接后，右击连接节点，选择“执行命令”选项，可以打开 SQL 命令窗口，实现对数据库的操作；选择“断开连接”选项，可以断开与数据库的连接。

在未建立与数据库连接的状况下，右击数据库连接节点，选择“删除”命令，可以删除数据库连接节点。

当需要建立与数据库的连接、而又不存在相应的连接节点时（如原有的连接点被删除，或一个数据库从另外一个计算机复制过来），可以按下面步骤创建一个数据库连接节点并建立与数据库的连接。

（1）右击“数据库”节点，在打开的快捷菜单中选择“新建连接”命令，打开“新建数据库连接”对话框。

（2）设置连接数据所需的相关属性，如图 11-3 所示。

图 11-3 “新建数据库连接”对话框

用于连接Java DB数据库的驱动程序分“Java DB(Embedded)”和“Java DB(Network)”两种。在Web应用中，一般应选择网络型驱动程序，即“Java DB(Network)”。

主机填Java DB数据库服务器所在机器的域名。若为本地，可填localhost。Java DB数据库服务器默认端口号为1527。

数据库名、用户名和口令应填在创建数据库时采用的相关属性。

选中“记住口令”复选框，让数据库资源管理器记住口令。这样，以后建立数据库连接时就不会被要求输入用户名和口令了。

(3) 单击“确定”按钮，在打开的“高级”选项卡中可以选择一个缺省架构。

若上述连接操作成功，一个数据库连接节点会被创建，同时自动建立从IDE至目标数据库的连接。

11.1.2 SQL语句

这里简单介绍Java DB中有关表的定义以及记录的插入、删除和更新等SQL语句。其中定义表的SQL语句分几步介绍。

1. 定义表的基本格式

定义表的SQL语句是CREATE TABLE，其基本格式如下：

```
CREATE TABLE [schema-name.]table-name (
    {column-definition|table-level-constraint}
    [,{column-definition|table-level-constraint}] *
)
```

架构名(schema-name)是可选项。若不指定架构名，将在当前缺省架构中创建表。

说明：

(1) 在SQL语法格式中，小写字体表示该项应根据需要和情况具体指定。

(2) 符号[]表示该项为可选项。

(3) 符号[] * 表示该项可重复0至多次。

(4) 符号|可以将两项或多项连接起来，表示选择其中一项。为标明第一项的开始处及最后一项的结尾处，可用符号{}将这些选项括起来。

2. 定义列

定义一个表的主要工作是定义表中各列，列定义(column-definition)的常用格式如下：

```
column-name data-type
    [column-level-constraint] *
    [{
        DEFAULT constant-expression|
        GENERATED {ALWAYS|BY DEFAULT} AS IDENTITY
            [(START WITH value1[,INCREMENT BY value2])]
    }]
    [column-level-constraint] *
```

列名(column-name)和列数据类型(data-type)是必选项。在Java DB中，常用的数据类型如表11-1所示。其中，第1列“SQL类型”给出的是在定义列时可采用的各种数据类型，

第 3 列“Java 类型”给出的是 Java 程序在处理数据表的各种类型数据时可采用的最适当的数据类型。

表 11-1　Java DB 数据库支持的数据类型

SQL 类型	说　　明	Java 类型
SMALLINT	2 字节短整型	java. lang. Short
INTEGER 或 INT	4 字节整型	java. lang. Integer
BIGINT	8 字节长整型	java. lang. Long
REAL	4 字节单精度浮点数	java. lang. Float
DOUBLE	8 字节双精度浮点数	java. lang. Double
DECIMAL[(precision[,scale])]	固定精度和小数位	java. math. BigDecimal
CHAR[(length)]	固定长度字符串，默认值长度为 1	java. lang. String
VARCHAR(length)	可变长度字符串，最多 32 672 字符	java. lang. String
DATE	日期	java. sql. Date
TIME	时间	java. sql. Time
TIMESTAMP	日期时间	java. sql. Timestamp
BLOB[(length[K|M|G])]	默认长度为 2G 字节(最大值)	java. sql. Blob
CLOB[(length[K|M|G])]	默认长度为 2G 字符(最大值)	java. sql. Clob

DEFAULT 短语可以为当前列指定缺省值，下面是一些例子：

- DEFAULT '男'
- DEFAULT 18
- DEFAULT DATE('2000-1-1')
- DEFAULT TIMESTAMP('2000-1-1 12:0:0')
- DEFAULT CURRENT_TIME
- DEFAULT CURRENT_TIMESTAMP

GENERATED 短语可以将当前列指定为身份列。一个表至多只能有一个身份列，身份列的数据类型只能是整数，包括 SMALLINT、INT 或 BIGINT。

身份列的值通常由系统自动产生，初值为 value1(默认值为 1)，以后每次增加 value2(默认值为 1)。身份列不会自动产生索引。

在定义身份列时，若指定 ALWAYS 选项，则身份列的值总是由系统自动产生，不能指定，也不能更新。在插入记录时，应缺省身份列，或将其值指定为 DEFAULT。此时，身份列的值是唯一的。

在定义身份列时，若指定 BY DEFAULT 选项，则身份列的值既可以自动产生，也可以显式指定。此时，身份列的值不能保证是唯一的。

3. 定义列级约束

在定义列时，可以指定列级约束(column-level-constraint)。列级约束的常用格式如下：

```
{
    NOT NULL|
    [CONSTRAINT constraint-name]
    {
        CHECK(logic-expression)|
        PRIMARY KEY|
        UNIQUE|
        REFERENCES [schema-name.]table-name[(column-name>)]
            [ON DELETE {NO ACTION|RESTRICT|CASCADE|SET NULL}]
            [ON UPDATE {NO ACTION|RESTRICT }]}
    }
}
```

NOT NULL 指明该列不能取空值。

其他列级约束可以有选择地指定一个约束名(constraint-name),包括:

(1) CHECK 约束指明该列取值应满足的条件,比如:CHECK (sex='男' OR sex='女')。

(2) PRIMARY KEY 指明该列是主关键字。主关键字不能是空值,会自动产生索引。

(3) UNIQUE 指明该列取值必须唯一。唯一键可以是空值,会自动产生索引。

(4) REFERENCES 指明该列是外键,并指出被参照的表和列。由于被参照列总是被参照表的主键列,所以被参照列通常可以被省略。

REFERENCES 约束同时可指明在当前表中有相关记录时是否可删除(DELETE)被参照记录或更新(UPDATE)被参照记录的主键。NO ACTION 或 RESTRICT 表示禁止删除被参照记录或更新其主键;CASCADE 表示在删除被参照记录时,同时删除当前表中的相关记录;SET NULL 表示在删除被参照记录时,将当前表中的相关记录的当前列设置为空值。

4. 定义表级约束

表级约束(table-level-constraint)往往涉及多个列,所以不能定义在单个列上,但表级约束与列级约束涉及的有关约束概念大致相同。定义表级约束的常用格式如下:

```
[CONSTRAINT <constraint-Name>]
{
    CHECK (<logic-expression>)|
    PRIMARY KEY(column-name[,column-name]*)|
    UNIQUE (column-name[,column-name]*)|
    FOREIGN KEY(column-name[,column-name]*)
        REFERENCES [schema-name.]table-name[(column-name[,column-name]*)]
        [ON DELETE {NO ACTION|RESTRICT|CASCADE|SET NULL}]
        [ON UPDATE {NO ACTION|RESTRICT }]
}
```

5. 定义表举例

这里以数据库 luntan 为例,说明表的定义。数据库 luntan 将应用于论坛应用项目,其中包含客户表、主题表和回复表,三个表都存放在该数据库的 loubuye 架构中。

客户表(client)的具体要求如表 11-2 所示,相应的定义语句如代码清单 11-1 所示。

表 11-2　客户表

列(字段)名	类　　型	说　　明
username	varchar(15)	登录名,主关键字
password	varchar(15)	口令,不能取空值
gender	char	性别,只能取"男"或"女",默认值为"男"
email	varchar(25)	邮件地址,不能取空值
degree	char	文化程度,可取空值(默认值)或以下各字符:1:高中或高中以下 2:大专 3:本科 4:硕士 5:博士或博士后

代码清单 11-1　定义 client 表

```
CREATE TABLE loubuye.client (
  username varchar(15) primary key,
  password varchar(15) NOT NULL,
  gender char NOT NULL CHECK(gender='男' OR gender='女') DEFAULT '男',
  email varchar(25) NOT NULL,
  degree char CHECK(degree IN('1','2','3','4','5'))
);
```

如果某列既没有指定 NOT NULL,也没有指定 DEFAULT 短语,则其默认值为空值。

主题表(topic)的具体要求如表 11-3 所示,相应的定义语句如代码清单 11-2 所示,假设当前缺省架构为 loubuye。

表 11-3　主题表

列(字段)名	类　　型	说　　明
id	int	主题帖编号,主关键字,身份列(初值和步长均为 1)
username	varchar(15)	发帖人,不能取空值,外键
title	varchar(50)	标题,不能取空值
content	varchar(1000)	内容,不能取空值
createtime	timestamp	发帖时间,不能取空值,默认值为当前日期时间
clickcount	int	点击数,不能取空值,默认值为 0
replycount	int	回复数,不能取空值,默认值为 0
lastreplyuser	varchar(15)	最后回复人,不能取空值,外键
lastreplytime	timestamp	最后回复时间,不能取空值,默认值为当前日期时间

代码清单 11-2　定义 topic 表

```
CREATE TABLE topic (
  id int PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  username varchar(15) NOT NULL CONSTRAINT topic_fk1 REFERENCES client,
  title varchar(50) NOT NULL,
```

```
    content varchar(1000) NOT NULL,
    createtime timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
    clickcount int NOT NULL DEFAULT 0,
    replycount int NOT NULL DEFAULT 0,
    lastreplyuser varchar(15) NOT NULL CONSTRAINT topic_fk2 REFERENCES client,
    lastreplytime timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

回复(跟帖)表(reply)的具体要求如表 11-4 所示,相应的定义语句如代码清单 11-3 所示,假设当前缺省架构为 loubuye。

表 11-4　跟帖表

列(字段)名	类　　型	说　　明
id	int	回复帖编号,主关键字,身份列(初值和步长均为 1)
username	varchar(15)	回复人,不能取空值,外键
content	varchar(1000)	回复内容,不能取空值
replytime	timestamp	回复时间,不能取空值,默认值为当前日期时间
topid	int	主题帖编号,不能取空值,外键

代码清单 11-3　定义 reply 表

```
CREATE TABLE reply (
  id int PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  username varchar(15) NOT NULL CONSTRAINT reply_fk1 REFERENCES client,
  content varchar(1000) NOT NULL,
  replytime timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  topid int NOT NULL CONSTRAINT reply_fk2 REFERENCES topic(id)
);
```

6. 修改表结构

修改表结构的语句是 ALTER TABLE,其基本格式如下:

```
ALTER TABLE table-Name
{
    ADD {COLUMN column-definition|table-level-constraint }|
    DROP { COLUMN column-name|CONSTRAINT constraint-name}|
    ALTER COLUMN column-Name
    {
        [ NOT ] NULL|
        SET DATA TYPE VARCHAR(integer)|
        RESTART WITH integer-constant|SET INCREMENT BY integer-constant|
        DEFAULT default-value|DROP DEFAULT
    }
}
```

该语句可以完成的功能包括：添加(ADD)新列或新的表级约束；删除(DROP)指定列或指定的约束；修改已有列的定义等。

对列的修改往往会有诸多限制，如某列已有记录取值为空值，就不能再将其设置为 NOT NULL。列的数据类型一般不能修改，但可以增加 VARCHAR 类型的宽度。

例如，要为客户表 client 添加一个状态列 status，列描述如下：

- 数据类型为 char(1)。
- 列取值范围：0(不在线)、1(在线)。
- 默认值为 0。

可以使用以下语句：

```
ALTER TABLE client ADD COLUMN status char CHECK(status IN('0','1')) DEFAULT '0';
```

又例如，要设置客户表 client 的状态列 status 不能取空值，可以使用以下语句：

```
ALTER TABLE client ALTER COLUMN status NOT NULL;
```

在执行该命令时，表中所有记录在 status 列上不能有空值。

7. 插入记录

插入记录的 SQL 语句是 INSERT，其基本格式如下：

```
INSERT INTO [schema-name.]table-name[(column-name[,column-name]* )]
    VALUES({expression|DEFAULT}[,{expression|DEFAULT}]*)
        [,({expression|DEFAULT}[,{expression|DEFAULT}]*)]*
```

一条 INSERT 语句可以插入一条记录或多条记录。表名(table-name)后括号内的列名表给出插入操作所涉及的列，各列的取值在 VALUES 短语中一一对应指定。关键字 DEFAULT 表示对应列取缺省值。对于列名表中没有给出的列，其值也取缺省值。如果缺省列名表，意味着包含表中所有列。

例如，下面语句可以在客户表 client 中插入一条记录：性别(gender)取缺省值“男”，文化程度(degree)取缺省空值，状态(status)取缺省值 0。

```
INSERT INTO client(username,password,email) VALUES('loubuye', '123456', 'loubuye@ 163.
com');
```

或

```
INSERT INTO client VALUES('loubuye', '123456', DEFAULT, 'loubuye@ 163.com',DEFAULT,
DEFAULT);
```

例如，下面语句可以在主题表 topic 中插入一条记录。

```
INSERT INTO topic VALUES(
    DEFAULT,                           /* 主题帖标识符自动产生 */
    'loubuye','title','content',       /* 发帖人、标题和内容不能取空值,也没有缺省值 */
    DEFAULT,DEFAULT,DEFAULT,'loubuye',DEFAULT);
```

例如，下面语句可以在主题表 topic 中插入多条记录。

```
INSERT INTO topic VALUES
    (DEFAULT,'loubuye','title1','content1',DEFAULT,DEFAULT,DEFAULT,'loubuye',DEFAULT),
    (DEFAULT,'loubuye','title2','content2',DEFAULT,DEFAULT,DEFAULT,'loubuye',DEFAULT),
    (DEFAULT,'loubuye','title3','content3',DEFAULT,DEFAULT,DEFAULT,'loubuye',DEFAULT);
```

8. 删除记录

删除记录的 SQL 语句是 DELETE,其常用格式如下:

```
DELETE FROM [schema-name.]table-name [WHERE boolean-expression]
```

WHERE 短语指定删除哪些记录,即用逻辑表达式(boolean-expression)指定删除条件。如果缺省 WHERE 短语,意味着删除表中所有记录。

例如,下面语句可以从主题表 topic 中删除表识符 id 为 1 的帖子。

```
DELETE FROM topic WHERE id=1;
```

9. 更新记录

更新记录的 SQL 语句是 UPDATE,其常用格式如下:

```
UPDATE [schema-name.]table-name
SET column-name={expression|DEFAULT} [,column-name={expression|DEFAULT}] *
[WHERE boolean-expression]
```

SET 短语指定为一列或多列设置新的值或缺省值(在定义表时指定的)。WHERE 短语指定更新哪些记录。如果缺省 WHERE 短语,意味着更新表中所有记录。

例如,下面可以更新主题表 topic 中编号为 2 的主题帖:

- 点击数和回帖数均置为 1。
- 最后回帖人设为“loubuye”。
- 最后回帖时间设为当前时间。

```
UPDATE topic
SET clickcount=1,replycount=1,
    lastreplyuser='loubuye',lastreplytime=CURRENT_TIMESTAMP
WHERE id=2;
```

11.2 JPA 概述

Java 持久性应用程序编程接口(Java Persistence API, JPA)为开发人员提供了一种实现对象—关系映射(Object-Relational Mapping, ORM)的设施。利用该设施,可以在应用程序中指定如何把一组 Java 类、对象和实例变量映射到关系数据库中的表、行和列,进而可以通过对 Java 对象的相关操作来实现对关系数据的创建、访问、更新和删除等操作。

在 JPA 中,相关 Java 类称为实体类。与普通的 Java 类相比,实体类的定义需要通过特定元数据定义它与关系数据库之间的映射关系。大多数情况下,一个实体类对应关系数据库中的一个表,一个实体类的实例对应表中的一条记录,一个持久性实例变量(属性)对应表中的一列(字段)。

JPA 包含三个方面内容：

(1) 对象一关系映射元数据，用于定义如何将实体类、变量映射到数据库表和列；

(2) EntityManager API，用于对实体执行 CRUD 持久化操作的标准 API；

(3) Java 持久性查询语言与 Query API，支持实体查询以及批量更新和删除操作。

JPA 本质上是一种标准或规范，不同的厂商可以有自己的实现。实现该规范的软件产品称为持久性提供器，常见的如 TopLink、Hibernate、EclipseLink 等。

说明：

(1) 术语"实体"有时指实体类、有时指实体类的实例或数据库中的记录，阅读时应根据上下文来理解。

(2) 实体类与关系数据库之间的映射关系既可以在 XML 文件中用特定元素进行声明，也可以直接在 Java 类中用特定 Java 标注进行声明。本书采用后一种方式。

11.3 实 体 类

通常，一个实体类代表关系数据库中的一个表。实体类的定义类似于一般 JavaBean 类的定义，但需要包含一些用于指定映射关系的 Java 标注。

下面是定义实体类时应满足的基本要求：

- 实体类必须由 Entity 型标注修饰，其主键实例变量必须由 Id 型标注或其他相应的标注修饰。
- 必须有一个 public 或 protected 的无参构造方法，也可以包含其他的构造方法。
- 实体类不能是 final 的，持久性实例变量的 getter 和 setter 方法也不能是 final 的。
- 持久性实例变量不能声明为 public，客户应该通过相关方法访问实体状态。
- 如果实体实例需要按值传递，那么实体类必须实现 java.io.Serializable 接口，即实体实例是可序列化的。

11.3.1 映射表

@Entity 标注用于修饰类，指定该类是一个实体类。@Table 标注用于修饰实体类，可以用 name 属性指定该实体类对应的数据库表的表名。@Table 标注是可选的。如果省略它，实体类映射至与实体类简单名同名的数据库表。

下面代码演示了如何将 Client 类映射至数据中的 CLIENT 表。

```
package entity;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Table;
@Entity
@Table(name="CLIENT")
public class Client implements Serializable {
    ......
}
```

因为数据库系统并不都支持混合大小写的表名称，所以在本例中，即使省略@Table 标注，持久化提供器通常也会将该实体类映射至 CLIENT 表。

如无特别说明，本章所述标注、接口等类型都定义在 javax. persistence 包中。

11.3.2 映射列

@Basic 标注指明一种基本映射，即其修饰的持久性实例变量值将直接映射为数据库中对应列的值。@Basic 标注可修饰以下类型的持久性实例变量：

- Java 基本类型、基本类型包装类。
- java. lang. String。
- java. math. BigInteger，java. math. BigDecimal。
- java. util. Date、java. util. Calendar、java. sql. Date、java. sql. Time、java. sql. TimeStamp。
- byte[]，Byte[]，char[]，Character[]。
- 枚举类型。
- 其他可序列化类型。

对这些类型的实例变量，即使没有显式指定@Basic 标注，也会隐式指定该标注，且其各属性取默认值。@Basic 标注的属性包括 optional 和 fetch。optional 属性指定该实例变量是否可以取 null，默认值为 true。fetch 属性指定实例变量值是延迟加载(FetchType. LAZY)还是预先获取(FetchType. EAGER)，默认值是 FetchType. EAGER。

默认情况下，实体类声明的每个持久性实例变量映射至相同名称的列，如 name 变量映射至 NAME 列。显式指定@Column 标注可以覆盖映射列的一些默认特性。下面代码指定实例变量 name 不能取 null 值、应映射到数据库表的 XM 列。

```
@Basic(optional=false)
@Column(name="XM")
private String name;
```

说明：在 JPA 中，映射列可以采用变量和属性两种方式。采用变量方式时，映射标注应该修饰持久性实例变量，持久性提供器将通过直接访问实例变量来保持实体实例的状态与数据库同步。采用属性方式时，映射标注应该修饰于持久性实例变量相应的 getter 方法，持久性提供器将通过调用 getter 方法和 setter 方法来保持实体实例的状态与数据库同步。为简化起见，本书仅采用变量方式。

当实例变量的类型为 java. util. Date 或 java. util. Calendar 时，需要显式使用@Temporal 标注以指明其将映射到数据库 DATE(包括年、月、日)、TIME(仅存储时间，不包括年、月、日)和 TIMESTAMP(包括年、月、日和时间)中的哪种数据类型。下面代码将实例变量 ordertime 映射至数据表列 ORDERTIME，其中实例变量的类型为 java. util. Date，数据表列的类型为 SQL 类型 TIMESTAMP。

```
@Column(name="ORDERTIME")
@Temporal(TemporalType.TIMESTAMP)
private Date ordertime;
```

@Temporal 标注只有唯一的 value 属性,其取值可以是:

```
• TemporalType.DATE                    //映射为日期
• TemporalType.TIME                    //映射为时间
• TemporalType.TIMESTAMP               //映射为日期时间
```

11.3.3 实体主键

每个实体类必须有一个主键,每个实体实例都有唯一的主键值。主键可以是单个持久性实例变量,此时该实例变量应该由 Id 型标注修饰。下面代码指明在实体类 Client 中,持久性实例变量 username 是主键。

```
public class Client implements Serializable {
    @Id
    private String username;
    ……
}
```

主键也可以由多个持久性实例变量组合而成,此时可以定义一个包含这些持久性变量的主键类,而原实体类的主键则可定义成该主键类类型的一个实例变量。如:

```
@Embeddable
public class OrderitemsPK implements Serializable {
    private int id;
    private String bh;
    ……
}
@Entity
public class Orderitems implements Serializable {
    @EmbeddedId
    protected OrderitemsPK orderitemsPK;
    ……
}
```

这里,每一个订单项由 id(订单号)和 bh(图书编号)共同标识。主键类用@Embeddable 标注修饰,而实体类中作为主键的主键类类型变量则用@EmbeddedId 标注修饰。

有些主键值需要由系统自动生成,这类主键称为代理键(surrogate key)。与之相对应,由业务数据构成的主键称为自然键(natural key)。代理键的例子如订单号、帖子编号等。

在 JPA 中,枚举类型 GenerationType 定义了代理键值自动生成的若干方式:

- GenerationType. IDENTITY:基于身份列生成。
- GenerationType. SEQUENCE:利用数据库序列生成。
- GenerationType. TABLE:利用序列表生成。
- GenerationType. AUTO:由持久性提供器自主选择某种方式生成。

在实体类中,代理键应该由@Id 标注和@GeneratedValue 标注同时修饰。代理键值的

自动生成方式可由@GeneratedValue标注的strategy属性指定。下面代码定义了一个代理键,键值采用身份方式自动生成。

```
@Entity
public class Topic implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;
    ......
}
```

这段代码表示TOPIC表中的ID列是一个主键,且是一个身份列。在基于身份列自动生成主键值时,在实体数据保存到数据库之前,其主键值可能不可用,因为通常在提交记录时才生成它。

如果ID列是一个主键(代理键),但不是一个身份列,那么可以用其他方式产生主键值。下面介绍如何利用序列表自动生成主键值。

首先应该定义用于生成键值的序列表,如:

```
CREATE TABLE sequence_generator_table (
    sequence_name varchar(15) PRIMARY KEY,
    sequence_value int NOT NULL
)
```

其中,sequence_name列用于存储序列的名称,sequence_value列用于存储序列的当前值。

接着需要在上述表中插入一条记录,指定某序列的序列名和当前值:

```
INSERT INTO sequence_generator_table (sequence_name,sequence_value)
                        VALUES('USER_SEQUENCE',0);
```

然后在实体类中,用@TableGenerator标注(基于序列表的主键值生成器)修饰类,设置主键值生成器的有关属性;用@GeneratedValue型标注修饰主键变量,指明用哪个生成器自动生成主键值。

```
@Entity
@TableGenerator(name="USER_GENERATOR",
    table="SEQUENCE_GENERATOR_TABLE",
    pkColumnName="SEQUENCE_NAME",
    valueColumnName="SEQUENCE_VALUE",
    pkColumnValue="USER_SEQUENCE",
    allocationSize=3)
public class Topic implements Serializable {
  @Id
  @GeneratedValue(strategy=GenerationType.TABLE,generator="USER_GENERATOR")
  private Integer id;
  ......
}
```

在@TableGenerator 标注中，name 属性是必选的，用于指定生成器的名称。其他属性的含义如下：

- table：指定序列表的表名。
- pkColumnName：序列表中存储序列名称的列名。
- valueColumnName：序列表中存储序列当前值的列名。
- pkColumnValue：指定序列名称。
- allocationSize：指定序列值的增量。

与基于身份列生成主键值不同，利用序列表生成主键值时，主键值在实体被持久操作时就会产生，而不是等到实体被实际同步至数据库时才产生。

11.3.4 关系映射

实体之间的关系包括一对一、一对多和多对多，其中最常用的是一对多和多对多关系。

1. 一对多关系

假设在数据库中包含一个客户(CLIENT)表和一个主题(TOPIC)表，其中客户和主题之间存在一对多和多对一关系，即每个客户可以有多个主题，而每个主题只能从属于某个客户。在数据库设计中，为体现这种一对多关系，往往在"多"端表(如主题表)中包含一个外键，存放"一"端表(如客户表)的主键值，如图 11-4 所示。

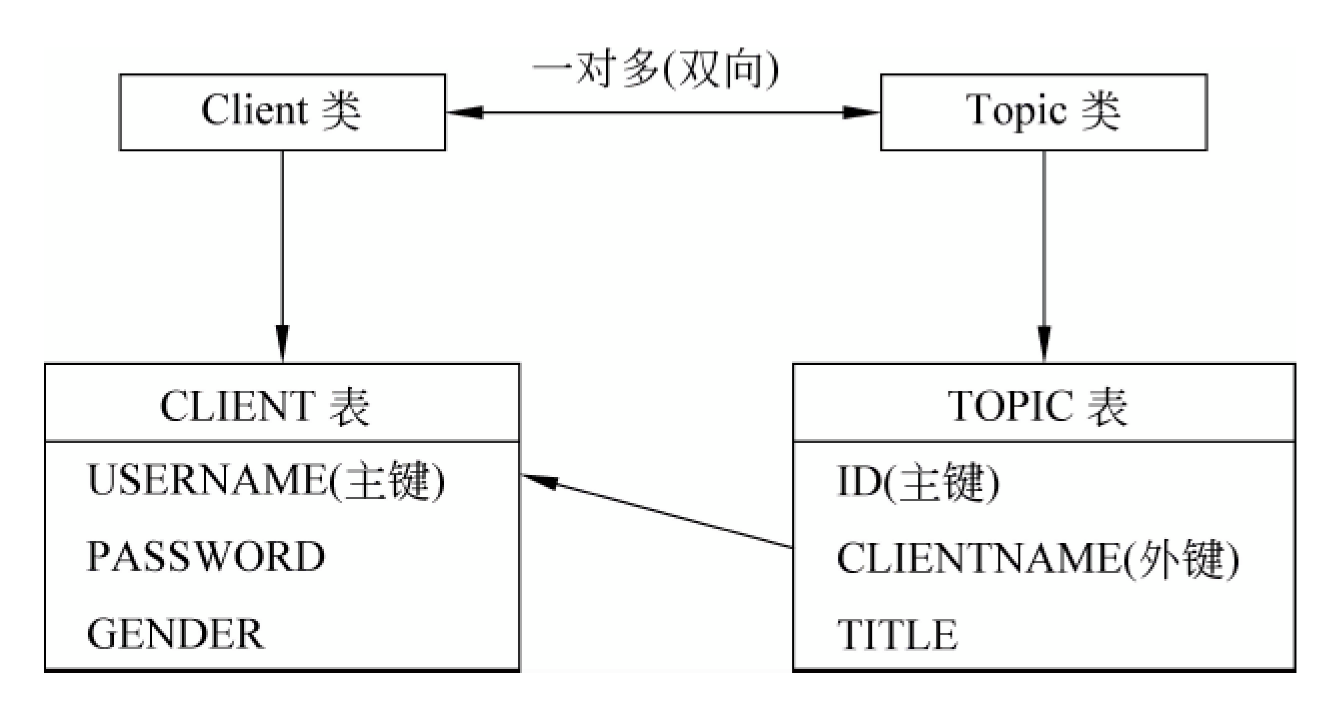


图 11-4　一对多关系

下面以此为例，介绍如何在实体类中定义实体之间的一对多和多对一的关系。这里把包含外键的一端称为关系的拥有端。首先定义关系拥有端实体类，即 Topic 类。

```
@Entity
@Table(name="TOPIC")
public class Topic implements Serializable{
    …
    @JoinColumn(name="CLIENTNAME",referencedColumnName="USERNAME")
    @ManyToOne
    private Client client;
    …
}
```

实体类 Topic 中定义了一个关系变量 client，其类型 Client 指明了关系另一端的实体类型。

@ManyToOne 标注指明多对一关系，即多个 Topic 型实体可以对应于一个 Client 实体。

@JoinColumn 标注指明形成关系的相关列，其中 name 属性指定关系拥有端实体类映射的数据库表中外键列的列名，referencedColumnName 属性指定关系非拥有端实体类映射的数据库表中主键的列名。

然后定义关系非拥有端实体类，即 Client 类。

```
@Entity
public class Client implements Serializable {
    …
    @OneToMany(mappedBy="client")
    private Collection<Topic>topicCollection;
    …
}
```

实体类 Client 中也定义了一个关系变量 topicCollection，其类型是集合(Collection)，其中泛型实参 Topic 指明了关系另一端的实体类型。

@OneToMany 标注指明一对多关系，即一个 Client 实体可以包含多个 Topic 型实体。该标注的 mappedBy 属性指定关系拥有端实体类中相应的关系变量。

如上定义所示，实体之间的一对多关系通常被定义成双向的，这使得查询工作变得非常简单。比如已通过 JPA 查询获得了一个主题对象，那么主题所属客户对象就会自动生成并保存在关系变量 client 中，调用相应的 getter 方法就能方便获得。反过来，如果已通过 JPA 查询获得了一个客户对象，那么该客户发表的所有主题对象也可以被自动生成并保存在关系变量 topicCollection 中，调用相应的 getter 方法可以容易地了解有关主题的信息。

2. 多对多关系

假设在数据库中包含一个雇员(EMPLOYEE)表和一个项目(PROJECT)表，其中雇员和项目存在多对多关系，即每个雇员可以参与多个项目，而每个项目也可以包含多个雇员。

在数据库设计中，为体现这种多对多关系，往往会创建一个关联表(E_P)，如图 11-5 所示。

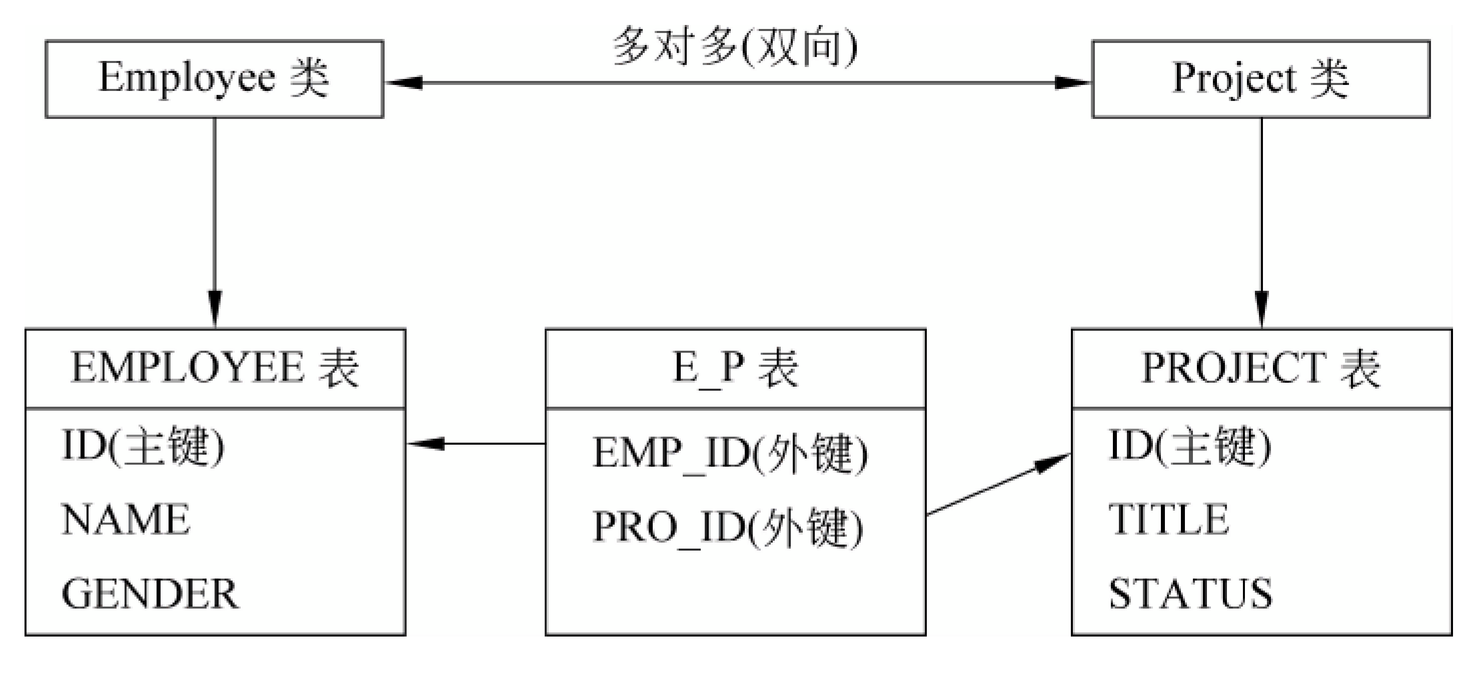


图 11-5　多对多关系

下面以此为例，介绍如何在实体类中定义实体之间的多对多关系。这里只需为雇员和项目分别定义实体类 Employee 和 Project，而无需为关联表定义相应的实体类。另外可以

任选一个实体类作为关系的拥有端实体类，这里假设 Employee 是关系的拥有端实体类。

```
@Entity
@Table(name="EMPLOYEE")
public class Employee implements Serializable {
    …
    @JoinTable(name="E_P",
        joinColumns=
            {@JoinColumn(name="EMP_ID",referencedColumnName="ID")},
        inverseJoinColumns=
            {@JoinColumn(name="PRO_ID",referencedColumnName="ID")}
    )
    @ManyToMany
    private Collection<Project>projectCollection;
    …
}
```

实体类 Employee 定义了一个关系变量 projectCollection，其类型是集合(Collection)，其中泛型实参 Project 指明了关系另一端的实体类型。

@ManyToMany 标注指明多对多关系，即 Employee 型实体与 Project 型实体之间存在多对多关系。

@JoinTable 标注指明关联表及相关列，其中 name 属性指定关联表的名称，joinColumns 属性指定关联表中的外键以及对应的在关系拥有端表中的主键，inverseJoinColumns 属性指定关联表中的外键以及对应的在关系非拥有端表中的主键。

然后定义关系非拥有端实体类，即 Project 类。

```
@Entity
@Table(name="PROJECT")
public class Project implements Serializable {
    …
    @ManyToMany(mappedBy="projectCollection")
    private Collection<Employee>employeeCollection;
    …
}
```

实体类 Project 定义了一个关系变量 employeeCollection，其类型是集合(Collection)，其中泛型实参 Employee 指明了关系另一端的实体类型。

@ManyToMany 标注指明多对多关系，即 Project 型实体与 Employee 型实体之间存在多对多关系。标注的 mappedBy 属性指定关系拥有端实体类中相应的关系变量。

11.4 通过数据库生成实体类

要通过数据库自动生成实体类，应该先创建数据库连接池以及相应的 JDBC 资源。下面以论坛项目为例，介绍如何通过论坛数据库自动生成相关实体类的过程。

11.4.1 创建数据库连接池

Java 应用程序在访问数据库之前需要首先建立与数据库的连接，然后利用连接对象完成对数据库的操纵。

数据库连接池(JDBC 连接池)是应用服务器(如 Glassfish)为特定数据库维护的一组可重用的连接。当应用程序需要访问数据库时，可以从连接池申请获得一个连接对象，而当访问结束后则可以将连接送回到连接池中。连接池通过提供可共享的数据库访问连接对象，可避免应用程序在每次访问数据库时都创建一个新的物理连接，从而缩短访问数据库的时间。

可以在 JSF 应用创建一个数据库连接池，其步骤如下。

(1) 在"项目"窗口中，选择 JSF 应用项目节点。

(2) 从"文件"菜单中选择"新建文件"命令。

(3) 选择文件类型：Glassfish|JDBC 连接池。

(4) 设置 JDBC 连接池的名称(luntanPool)，然后选中"从现有连接中提取"单选钮并从下拉列表中选择相应的数据库连接，如图 11-6 所示。



图 11-6 创建 JDBC 连接池

(5) 单击"下一步"按钮，进入"添加连接池属性"选项卡。一些基本属性已经从前面选定的数据库连接中提取。要添加新的属性可单击"添加"按钮。

(6) 单击"下一步"按钮，进入"指定连接池的可选属性"选项卡。这里，可以设置连接池的一些其他属性，如：连接池的最大数量、空闲间隔时间等。

(7) 单击"完成"按钮。

在使用"新建文件"向导创建项目的 JDBC 连接池后，连接池的名称和相关属性会被保存在 sun-resources.xml 文件中，该文件出现在项目的"服务器资源"节点下。可以在源代码编辑器中编辑 sun-resources.xml 以修改相关属性。

11.4.2 创建 JDBC 资源

通常，Java 应用程序并不能直接向 JDBC 连接池中申请连接对象，而是通过 JDBC 资源

来获得连接对象。

JDBC 资源又称数据源，是对 JDBC 连接池的一个包装。每个 JDBC 资源都有一个 JNDI 名称。Java 应用通过该名称获得 JDBC 资源，进而获得连接对象并访问数据库。

可以在 JSF 应用创建一个 JDBC 资源，其步骤如下。

(1) 在“项目”窗口中，选择 JSF 应用项目节点。

(2) 从“文件”菜单中选择“新建文件”命令。

(3) 选择文件类型：GlassFish|JDBC 资源。

(4) 在“常规属性”选项卡中，选择现有连接池，或创建一个新的连接池。需要时修改该资源的 JNDI 名称。这里，选择现有的连接池 luntanPool，并指定 JNDI 名称为 jdbc/luntan，如图 11-7 所示。



图 11-7　创建 JDBC 资源

(5) 单击“下一步”按钮，进入“附加属性”选项卡。需要时可以添加该资源的附加属性。

(6) 单击“完成”按钮。

创建项目的 JDBC 资源后，该资源的 JNDI 名称和相关属性也被保存在项目的“服务器资源”节点下的 sun-resources.xml 文件中。

当部署 JSF 应用项目，定义在项目中 JDBC 连接池和 JDBC 资源将自动在应用服务器中进行配置和注册。这些资源既可以被当前应用项目使用，也能被其他的应用项目使用。

11.4.3　生成实体类

有了 JDBC 资源，就可以利用工具通过数据库自动生成实体类了，步骤如下。

(1) 在“项目”窗口中，选择 JSF 应用项目节点。

(2) 从“文件”菜单中选择“新建文件”命令。

(3) 选择文件类型：持久性|通过数据库生成实体类。

(4) 指定数据源，即 JDBC 资源。然后从左边的“可用表”中选择需要生成实体类的相关表，IDE 将自动为右边列表中列出的每个表生成实体类。这里，选择数据源为 jdbc/

luntan，并单击“全部添加”按钮选择所有表，如图 11-8 所示。



图 11-8 指定要生成实体类的数据库表

(5) 单击“下一步”按钮，进入“实体类”选项卡，确认实体类的名称、位置。实体类通常存放在单独的 Java 包中，如事先还没有创建相应的 Java 包，可以在此直接指定包名，IDE 将自动创建之。这里，指定 entity 包用于存放生成的实体类，如图 11-9 所示。



图 11-9 指定存放实体类的 Java 包

默认情况下，IDE 会为实体类自动生成针对持久性变量的一些命名查询标注，另外也会自动创建相应的持久性单元。有关命名查询和持久性单元的详细内容将在第 12 章介绍。

(6) 单击“下一步”按钮，进入“映射选项”选项卡，可以根据需要指定从实体类到表的一些映射选项。

(7) 单击“完成”按钮。

11.5 论坛—创建数据库

本节继续第 6.5 节介绍的应用项目(luntan_pagination)，为论坛应用创建所需的 Java DB 数据库，并产生相应的实体类。

本章前面已介绍了 Java DB 的基本操作、SQL 语句、实体类以及通过数据库自动生成实体类等内容，而所举例子也是以论坛数据库为主，所以本节主要是对前面内容进行一个总结，列出创建论坛数据库和产生相应实体类的总体步骤。

为保持之前项目的独立性，这里新创建一个名为 luntan_final 的 JSF 应用项目，并从原先项目复制所有的样式表文件、JSF 页面、源包中的所有 Java 包和 Java 类。

11.5.1 创建论坛数据库

在"服务"窗口，按以下步骤创建论坛数据库。

(1) 根据实际情况设置存放数据库的位置。

(2) 创建论坛数据库，指定数据库名称、用户名和口令。为后面叙述方便，假设数据库名为 luntan，用户名为 loubuye。

(3) 数据库创建后，在"数据库"节点下会自动产生一个连接节点。该连接节点的缺省架构名字为创建数据库时指定的用户名。如果数据库中不存在该架构，则应在 SQL 命令窗口中用命令"CREATE SCHEMA loubuye"创建之。

要查看数据库中是否存在某架构，可右击数据库连接节点，然后选择"连接"命令，建立与数据库的实际连接。此时，连接节点下会显示出数据库中的所有架构。

要打开 SQL 命令窗口，可右击数据库连接节点，然后选择"执行命令"命令。执行完 SQL 语句后，可右击连接节点，然后选择"刷新"命令，以查看执行效果。

(4) 在 SQL 命令窗口，用代码清单 11-1 所示 SQL 语句创建客户数据库表(client)。

(5) 为客户表 client 添加一个状态列 status，列描述如下：

- 数据类型为 char(1)。
- 列取值范围：0(不在线)、1(在线)。
- 默认值为 0。
- 不能取空值。

(6) 在 SQL 命令窗口，用代码清单 11-2 所示 SQL 语句创建主题数据库表(topic)。

(7) 在 SQL 命令窗口，用代码清单 11-3 所示 SQL 语句创建回复数据库表(reply)。

(8) 在 SQL 命令窗口，用 INSERT INTO 语句往客户表中添加几条记录，再往主题表中添加几条记录。

11.5.2 为论坛应用创建实体类

在"项目"窗口，通过数据库为论坛应用自动生成实体类，具体步骤如下。

(1) 在论坛应用 luntan_final 中，基于现有连接，为论坛数据库 luntan 创建 JDBC 连接池，假设连接池的名称指定为 luntanPool。

创建 JDBC 连接池后，其名称和相关属性被保存在 sun-resources. xml 文件中，该文件放置在项目的"服务器资源"节点下。

(2) 在论坛应用 luntan_final 中，基于以上建立的 JDBC 连接池，创建 JDBC 资源，假设该资源的 JNDI 名称指定为 jdbc/luntan。

与 JDBC 连接池一样，创建 JDBC 资源后，该资源的 JNDI 名称和相关属性也被保存在

sun-resources.xml 文件中。

(3) 在论坛应用 luntan_final 中，基于以上的 JDBC 资源(数据源)，为论坛数据库的所有数据库表生成相应的实体类。保存实体类的位置选择源包中的 entity 包，这样生成的实体类将覆盖 entity 包中原先的 Client 类、Topic 类和 Reply 类。

代码清单 11-4 是通过数据库自动生成的 Client 实体类的部分代码，其中省略了那些不含映射标注的方法代码。

代码清单 11-4　Client 实体类(部分)

```
package entity;
import java.io.Serializable;
import java.util.Collection;
import javax.persistence.Basic;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name="CLIENT")
@NamedQueries({
  @NamedQuery(name="Client.findAll",query="SELECT c FROM Client c"),
  @NamedQuery(name="Client.findByUsername",
              query="SELECT c FROM Client c WHERE c.username=:username"),
  @NamedQuery(name="Client.findByPassword",
              query="SELECT c FROM Client c WHERE c.password=:password"),
  @NamedQuery(name="Client.findByGender",
              query="SELECT c FROM Client c WHERE c.gender=:gender"),
  @NamedQuery(name="Client.findByEmail",
              query="SELECT c FROM Client c WHERE c.email=:email"),
  @NamedQuery(name="Client.findByDegree",
              query="SELECT c FROM Client c WHERE c.degree=:degree"),
  @NamedQuery(name="Client.findByStatus",
              query="SELECT c FROM Client c WHERE c.status=:status")})
public class Client implements Serializable {
  private static final long serialVersionUID=1L;
  @Id
  @Basic(optional=false)
  @Column(name="USERNAME")
  private String username;
```

```
    @Basic(optional=false)
    @Column(name="PASSWORD")
    private String password;
    @Basic(optional=false)
    @Column(name="GENDER")
    private char gender;
    @Basic(optional=false)
    @Column(name="EMAIL")
    private String email;
    @Column(name="DEGREE")
    private Character degree;
    @Basic(optional=false)
    @Column(name="STATUS")
    private char status;
    @OneToMany(cascade=CascadeType.ALL,mappedBy="client")
    private Collection<Reply> replyCollection;
    @OneToMany(cascade=CascadeType.ALL,mappedBy="client")
    private Collection<Topic> topicCollection;
    @OneToMany(cascade=CascadeType.ALL,mappedBy="client1")
    private Collection<Topic> topicCollection1;

    public Client(){
    }
    public Client(String username){
      this.username=username;
    }
    public Client(String username,String password,char gender,String email,char status){
      this.username=username;
      this.password=password;
      this.gender=gender;
      this.email=email;
      this.status=status;
    }
    ...
}
```

与原先的 Topic 类和 Reply 类相比，新生成的 Topic 实体类和 Reply 实体类并没有实现 Comparable 接口、提供 compareTo 方法，也就是说，两个 Topic 对象或两个 Reply 对象是不能进行相互比较的。这样，就不能用 Collections 类的 sort 方法对一个 List＜Topic＞表中的元素或一个 List＜Reply＞中的元素进行排序。因此，当按上述步骤生成实体类后，项目中会出现两个错误。一个是 model. TopicManager 类的 getTopics() 方法中的语句：

```
Collections.sort(topics);
```

另一个是 model.ReplyManager 类的 getReply(int)方法中的语句:

```
Collections.sort(list);
```

只要将上面两条语句删除,应用项目就能够正常运行了。当然,此时不能保证页面上主题表中各主题以及回复表中各回复是按一定的时间顺序排序的。

另外,虽然在应用中已创建了实体类,但 model.TopicManager、model.ReplyManager 等类中的业务方法访问的仍然是定义在 model.DataBase 类中的数据。

但这些都不是问题,在第 12 章,我们将重写这些业务方法,让应用真正从已经创建的 luntan 数据库中访问数据。到那时,获取按一定的顺序排序的主题记录或回复记录也就变成一件轻而易举的事情了。

11.6 小　　结

- Java DB 是一个纯 Java 实现、开源的数据库管理系统,是 Java SE 6 平台的组成部分, NetBeans IDE 及 Glassfish 同样提供了对 Java DB 的支持。
- 一个 Java DB 数据库包含若干架构,每个架构下包含若干数据库表。数据库连接节点在创建时指定一个缺省架构。
- Java 持久性应用程序编程接口(Java Persistence API, JPA)包含三个方面内容:
  对象-关系映射元数据;
  EntityManager API;
  Java 持久性查询语言与 Query API。
- 实体类的定义类似于一般 JavaBean 类的定义,但还需要有特别的映射定义,即要定义实体类—数据库表、实例变量—字段等的映射关系。
- JDBC 连接池是应用服务器为特定数据库维护的一组可重用的连接,能为各应用共享,可为各应用缩短连接数据库所需的时间。
- JDBC 资源又称数据源,是对 JDBC 连接池的一个包装,为本地或远程的各种应用提供了一个连接和访问数据库的统一的 JNDI 名称。
- 在 NetBeans IDE 中,可以通过已有的数据库自动生成相应的实体类。

习　题　11

1. 用于连接 Java DB 数据库的驱动程序有哪几种? 在 Web 应用中,一般应该选择哪种驱动程序?

2. 在 NetBeans IDE 中,当创建一个 Java DB 数据库时,会自动产生一个连接节点。在该连接节点中,数据库的缺省架构的名称是什么?

3. 与普通的 Java 类或 JavaBean 类相比,实体类有什么特点?

4. 创建一个 Java DB 数据库:数据库名称为 bookstore、用户名为 sample、密码为 123456,然后再完成:

（1）在该数据库的 sample 架构下创建数据库表 CLIENT，具体要求如表 11-5 所示。

表 11-5　bookstore 数据库的 CLIENT 表

列(字段)名	类型与宽度	说　　明	列(字段)名	类型与宽度	说　　明
username	varchar(10)	用户名，非空、关键字	phone	varchar(15)	
password	varchar(10)	口令，非空	email	varchar(25)	非空
realname	varchar(4)	真实姓名，非空	address	varchar(30)	
sex	char(1)	性别，非空			

（2）在该数据库的 sample 架构下创建数据库表 BOOK，具体要求如表 11-6 所示。

表 11-6　bookstore 数据库的 BOOK 表

列(字段)名	类型与宽度	说　　明	列(字段)名	类型与宽度	说　　明
bookid	char(6)	编号，非空、关键字	banci	varchar(12)	版次
title	varchar(20)	书名，非空	yinci	varchar(14)	印次
author	varchar(4)	作者，非空	kaiben	varchar(15)	开本
publisher	varchar(10)	出版社，非空	yeshu	varchar(6)	页数
price	numeric(6,2)	定价，非空	zishu	varchar(6)	字数
isbn	varchar(15)	ISBN 号，非空	zhuzhen	varchar(4)	装帧

（3）利用 SQL INSERT 语句在 CLIENT 表中添加 2 至 3 条记录，在 BOOK 表中添加 10 条以上记录。

第 12 章　实体管理器与 JPQL

本章主题：

- 持久性单元
- 实体管理器与持久性上下文
- 实体状态与 CRUD 操作
- JPQL 语言
- 查询 API

每个实体管理器都与一个持久性上下文相关联，持久性上下文管理着一组实体实例，并在适当时机使其与底层数据库同步。持久性上下文能够管理的实体实例的类型由持久性单元定义。利用实体管理器可以创建、更新、删除实体实例，或根据主键检索实体实例，进而实现对数据库数据的相应操作。

JPQL(Java Persistence Query Language，Java 持久性查询语言)可以实现对实体实例的批量查询、更新和删除。JPQL 在语法上与 SQL 相似，但并不能在数据库系统上直接执行。要执行 JPQL 语句，需要先创建包装有 JPQL 语句的查询(Query)对象，然后通过调用该查询对象的相关方法执行其中的 JPQL 语句。

12.1　持久性单元

要在应用程序中通过操作实体实现数据库访问，首先需要定义持久性单元。一个持久性单元通常需要定义下面一些属性：

- 一组实体类。
- 一个数据源。
- 映射元数据(Java 标注或 XML 文件)。
- 持久性提供器(实体管理器工厂和实体管理器)。

可见，一个持久性单元定义了持久性操作的对象范围，即需要管理哪些实体，包括实体类、数据源以及两者之间的映射。另外，持久性单元也定义了进行持久性操作的工具，即持久性提供器，其中包括实体管理器工厂和实体管理器。

持久性单元由 persistence.xml 文件定义，包含持久性单元的相关属性。persistence.xml 文件中可以定义多个持久性单元，但每个持久性单元必须有唯一的名称。persistence.xml 文件应该存放在 META-INF 目录下，而存放 META-INF 目录的父目录被称为持久性单元的根目录。对于 Web 应用来说，持久性单元的根目录通常是 WEB-INF/classes。在“项目”窗口，persistence.xml 文件位于“配置文件”节点下。

下面以论坛项目为例，介绍定义持久性单元的步骤。

(1) 在“项目”窗口中，选择 JSF 应用项目节点。

(2) 从“文件”菜单中选择“新建文件”命令。

(3) 选择文件类型：持久性|持久性单元。

(4) 单击“下一步”按钮，进入“提供器和数据库”选项卡，指定持久性单元的相关属性，包括持久性提供器和数据源。持久性提供器提供实体管理和持久性操作的功能，数据源指定与特定数据库的连接信息。

在这里，把持久性单元名称设置为“luntanPU”，持久性提供器选择为缺省的“EclipseLink(JPA 2.0)”，数据源指定为“jdbc/luntan”。另外取消“使用 Java 事务 API”复选框，意味着选用本地资源事务。将表生成策略指定为“无”，表明数据库表不由实体类的生成而生成、删除而删除，如图 12-1 所示。



图 12-1 指定提供器和数据源

(5) 单击“完成”按钮。

其中，“jdbc/luntan”是在 11.5.2 小节中创建的 JDBC 资源的 JNDI 名称。利用该资源可以实现与论坛数据库的连接。

如果是第一次定义持久性单元，工具将在相应位置自动创建 persistence.xml 文件，持久性单元的相关属性将保存在该文件。如果是再次定义持久性单元，新的持久性单元的相关属性同样保存在该文件中。代码清单 12-1 是 persistence.xml 文件的一个例子。

代码清单 12-1 persistence.xml 文件示例

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
 <persistence-unit name="luntanPU" transaction-type="RESOURCE_LOCAL">
   <non-jta-data-source>jdbc/luntan</non-jta-data-source>
   <exclude-unlisted-classes>false</exclude-unlisted-classes>
   <properties/>
 </persistence-unit>
</persistence>
```

这个 persistence.xml 文件仅定义了一个持久性单元 luntanPU，其中 non-jta-data-source 元素指定了数据源；exclude-unlisted-classes 元素指定了实体类，即该持久性单元包

含项目中所有的实体类。另外，由于选择的持久性提供器是缺省的 EclipseLink(JPA 2.0)，所以在该持久性单元的定义中并没有用 provider 元素明确指定。

实际上，在“通过数据库生成实体类”时，如果项目中还没有定义任何持久性单元，IDE 会自动创建一个持久性单元，只是自动创建的持久性单元与上述有所不同，比如没有指定实体类。此时，可以在原先定义的基础上，按上述要求进行修改，或者干脆删除原先的持久性单元定义或文件，再按上述步骤重新创建。

12.2 管理实体

本节首先介绍实体管理器和持久性上下文的概念、实体管理器 API 的基本情况，然后详细介绍操作实体实例的各种方法。

11.2.1 实体管理器与持久性上下文

实体由实体管理器(EntityManager 对象)管理。每个实体管理器与一个持久化上下文相关联。持久化上下文是一组受管理实体实例的集合。实体管理器借助持久化上下文实现对实体的管理。持久化上下文保证受管理实体实例与底层数据库的同步，如图 12-2 所示。

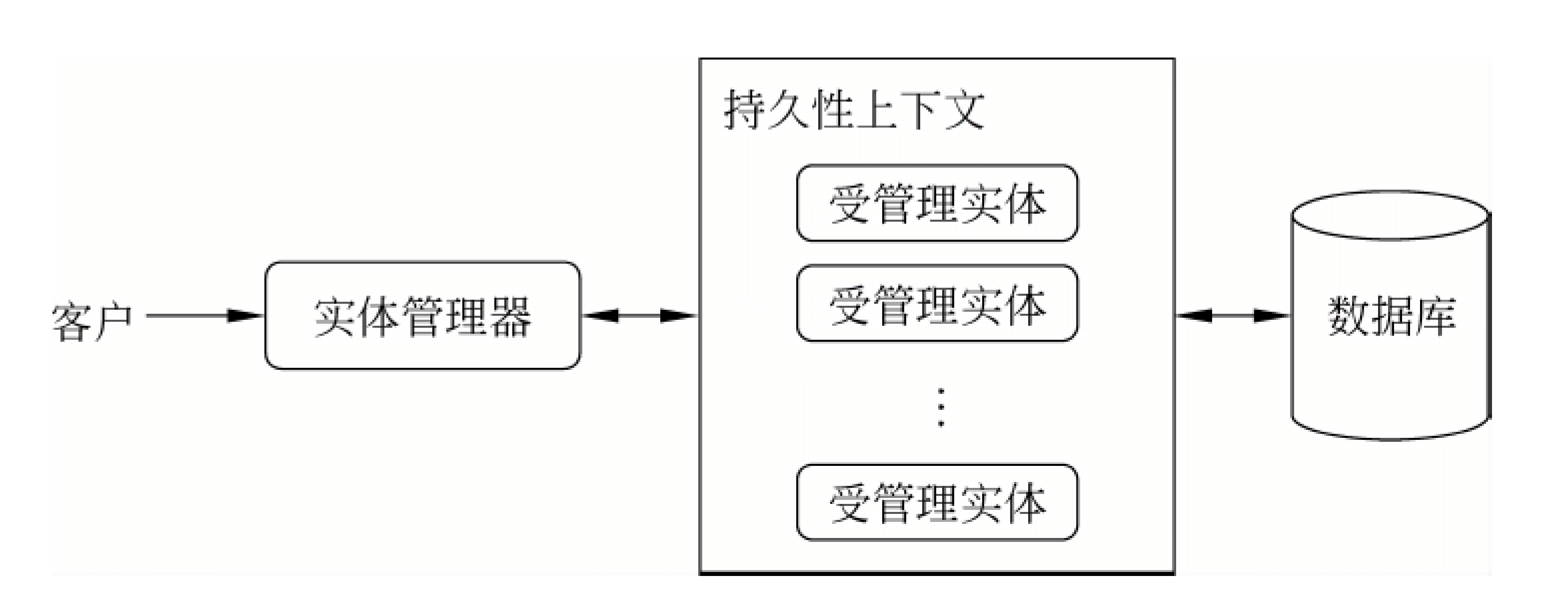


图 12-2　实体管理器与持久性上下文

要创建一个实体管理器，可以先获得一个实体管理器工厂对象。调用 Persistence 类的 createEntityManagerFactory 静态方法可以获得一个实体管理器工厂，其中应指定持久性单元的名称。有了实体管理器工厂，再调用其 createEntityManager 方法就可创建一个应用程序管理的实体管理器。

```
EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
EntityManager em=emf.createEntityManager();
```

调用实体管理器的相关方法可以完成对实体的创建、读取、更新和删除(Create、Read、Update、Delete，CRUD)等操作。表 12-1 列出实体管理器的一些主要方法。

close 方法用于关闭实体管理器并释放相关资源。一旦执行了该方法，就不应再试图调用该实体管理器的方法进行相关的实体操作，或通过由该实体管理器创建的查询对象执行查询操作。

每个实体管理器都会与一个持久性上下文相关联。与应用程序管理的实体管理器相关联的持久性上下文会随着实体管理器的创建而自动生成，通常也会随着实体管理器的关闭

而自动释放。

表 12-1　EntityManager 接口中的常用方法

方　　法	描　　述
void persist(Object entity)	使指定实体实例持久性并受管理
<T> T merge(T entity)	合并指定实体实例状态至持久性上下文
void remove(Object entity)	删除指定的实体实例
<T> T find(Class<T> entityClass, Object key)	根据主键值查找实体实例
void flush()	用持久性上下文中的实体实例状态刷新底层数据库
void refresh(Object entity)	用底层数据库重置指定实体实例的状态
Query createQuery(String jpqlString)	根据 JPQL 语句创建查询对象
Query createNamedQuery(String name)	创建查询对象以执行命名查询
void close()	关闭应用程序管理实体管理器
void clear()	使所有受管理实体实例成为分离的
EntityTransaction getTransaction()	获得资源本地类型的事务对象

12.2.2　实体操作

这里介绍若干常用的用于操作实体实例的 EntityManager 方法。通过这些方法，我们可以进行对实体实例的 CRUD 操作，进而完成对关系数据的创建、读取、更新和删除操作。

1. 实体的状态

EntityManager 方法可能会改变实体实例的状态。实体实例的状态包括四种：新的、受管理的、分离的和删除的。

(1) 新的实体实例：没有持久性身份、还没有与持久性上下文相关联。

(2) 受管理的实体实例：有持久性身份、与持久性上下文相关联。

(3) 分离的实体实例：有持久性身份、当前没有与持久性上下文相关联。

(4) 删除的实体实例：有持久性身份、与持久性上下文相关联，但其映射的持久化存储数据将被删除。

图 12-3 总结了实体实例的状态转换。

与持久性上下文相关联的实体实例包括受管理和删除的实体实例。对这些实体实例，持久性提供器会在适当的时机保证它们与数据库同步，比如用受管理实体实例的状态更新数据库中相应记录的数据，从数据库中删除与删除的实体实例相对应的记录。利用实体管理器的 remove 方法可以将受管理的实体实例转换成删除的实体实例。

新的实体实例没有持久性身份，意味着它在数据库中还没有相应的记录。利用实体管理器的 persist 方法可以将新的实体实例转换成受管理的实体实例。这样，持久性提供器在确保实体数据与数据库同步时，就会在数据库中插入相应的记录。

分离的实体实例有持久性身份，意味着它在数据库中有相应的记录。分离的实体实例没有与持久性上下文相关联，所有分离的实体实例状态的改变并不能同步至数据库。利用

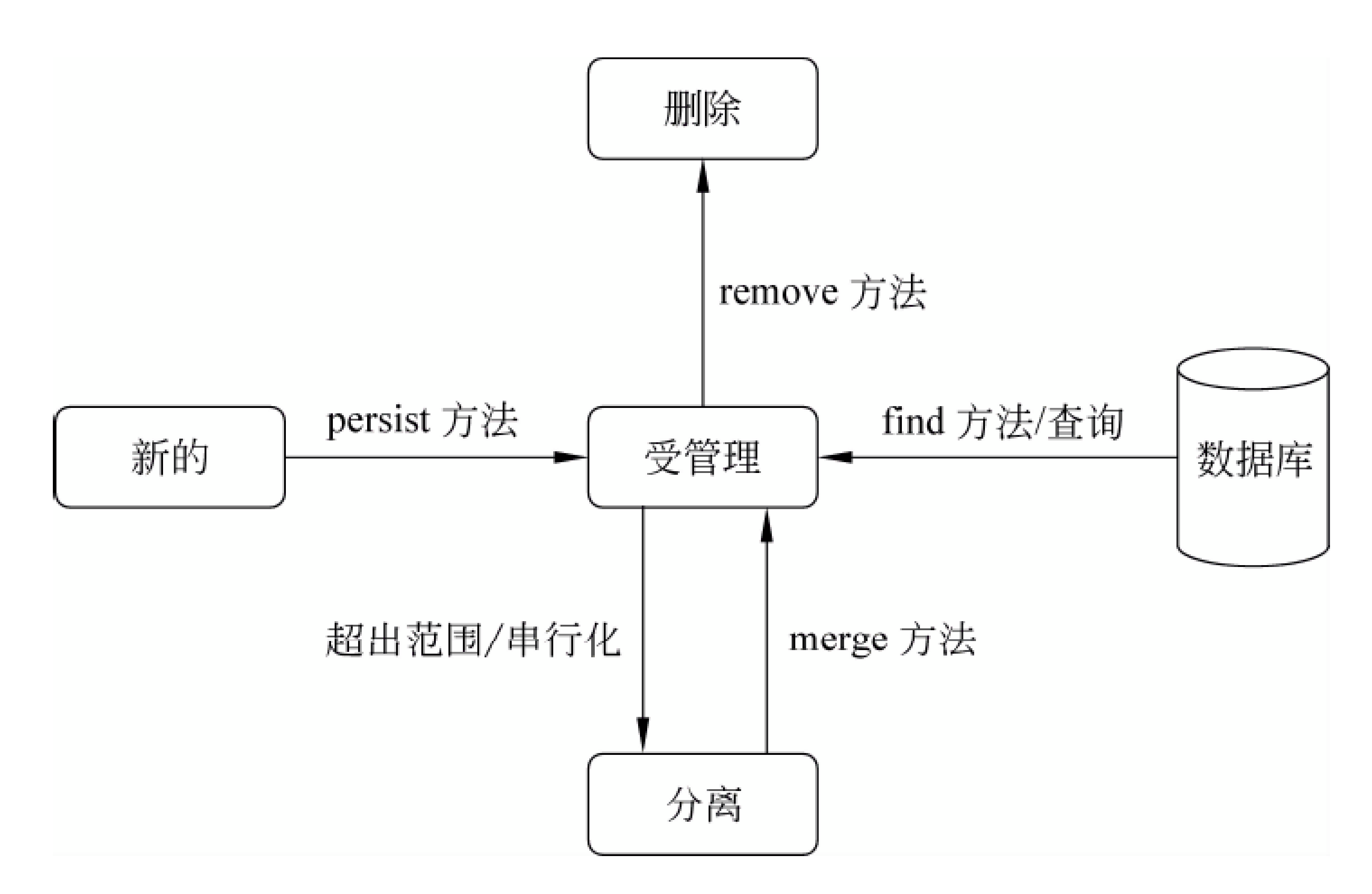


图 12-3　实体状态转换图

实体管理器的 merge 方法可以将分离的实体实例转换成受管理的实体实例。

2. 查找实体实例

查找实体实例的最简单方法是使用 EntityManager 的 find 方法。其他方法涉及 JPQL 和查询 API，这些内容将在 12.4 节详细介绍。

EntityManager 的 find 方法能够根据主键值查找相应的实体实例，其格式如下：

```
public <T>T find(Class<T>entityClass,Object primaryKey)
```

使用说明：

(1) 第一个参数的类型必须是实体类型；第二个参数的类型必须是实体的主键类型。

(2) 若方法找到相应的实体实例，就返回该实体实例，且该实体实例处于受管理状态；否则返回 null。

(3) 若第一个参数不是一个实体类型或第二个参数类型不是相应实体的有效主键类型，方法将抛出 java.lang.IllegalArgumentException 型例外。

代码清单 12-2 演示了 find 方法的使用。其中 luntanPU 是持久性单元的名称，Client 是一个实体类。用户名(username)是 Client 实体类的主键。findClientByName 方法接收客户的用户名，若数据库中存在相应的客户记录或者持久性上下文中有相应的 Client 实例，方法返回相应的 Client 实例，否则方法返回 null。

代码清单 12-2　find 方法使用示例

```
public Client findClientByName(String username){
  EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
  EntityManager em=emf.createEntityManager();
  Client client=em.find(Client.class,username);
  em.close();
  emf.close();
  return client;
}
```

从数据库查找(find)或查询到的实体包含一些列的持久性数据，如一个客户的密码、性

别以及他发表的主题等。这些数据的获取存在两种模式：预先获取(eager fetch)和延迟获取(lazy fetch)。预先获取是指在检索或查询实体时即加载相应数据；延迟获取是指不在检索或查询时即时加载，而是在需要(被访问)时再加载相应数据。枚举类型 FetchType 定义了这两种获取模式：

- FetchType. EAGER。
- FetchType. LAZY。

第 11 章曾介绍，实体类中反映基本映射的持久性变量通常用@Basic 标注修饰，该标注的 fetch 属性可以指定持久性变量值的获取模式，默认情况下取 FetchType. EAGER。

除了@Basic 标注，用于修饰关系变量的一些标注也包含 fetch 属性。利用 fetch 属性可以指定这些关系变量的获取模式。其中，@OneToOne 和@ManyToOne 标注中 fetch 属性的默认值为 FetchType. EAGER，而@OneToMany 和@ManyToMany 标注中 fetch 属性的默认值为 FetchType. LAZY。也就是说在默认情况下，当检索到的一个实体包含单个相关实体时，就预先加载该相关实体，当检索到的一个实体可能包含多个相关实体时，就延迟加载这些相关实体。

3. 持久实体实例

要插入新的数据记录可以调用 EntityManager 的 persist 方法。persist 方法使一个新的实体实例成为受管理的。受管理的实体实例会在所在事务递交前或执行 flush 操作时保存到数据库中。persist 方法的方法签名如下：

```
public void persist(Object entity)
```

使用说明：

(1) 调用方法时必须指定一个实体实例，否则将抛出 java. lang. IllegalArgumentException 型例外。

(2) 若指定的实体实例原先就是受管理的，则该操作被忽略，即实体实例仍保持受管理的。

(3) 若指定的实体实例原先是删除的，那么该实体实例将成为受管理的。

(4) 若指定的实体实例是分离的，方法将抛出 EntityExistsException 型例外，或者在事务递交或执行 flush 操作时，抛出 EntityExistsException 型例外或其他 PersistenceException 型例外。

PersistenceException 是持久性提供器抛出的各种例外类型的超例外类型，该例外类型与 java. lang. IllegalArgumentException 一样，都是运行时例外。

代码清单 12-3 演示了 persist 方法的使用。insertClient 方法接收一个 Client 型实体实例，通过 persist 方法将其纳入持久性上下文，并在事务递交时在数据库中插入相应的记录。若 persist 方法没有抛出例外，比如方法接收的是一个新的实体实例且各持久性数据都符合要求，方法返回 true；否则，方法返回 false。方法中涉及的有关事务处理的内容会在 12.3 节做进一步介绍。

代码清单 12-3　persist 方法使用示例

```
public boolean insertClient(Client client){
  EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
```

```
EntityManager em=emf.createEntityManager();
boolean success=true;
try {
  em.getTransaction().begin();
  em.persist(client);
  em.getTransaction().commit();
} catch(PersistenceException e){
   success=false;
}
em.close();
emf.close();
return success;
}
```

若进行持久操作的实体实例(包括受管理的实体实例)存在相关的实体实例,且关系标注(如@OneToMany 等)的 cascade 属性被设置为 PERSIST 或 ALL,则进行层叠持久操作,即相关的实体实例也进行持久操作。

层叠操作的类型在枚举类型 CascadeType 中定义,共包含 6 个枚举常量:

- CascadeType. ALL:把对实体的所有操作传播到相关实体。
- CascadeType. PERSIST:把对实体的 persist 操作传播到相关实体。
- CascadeType. REMOVE:把对实体的 remove 操作传播到相关实体。
- CascadeType. MERGE:把对实体的 merge 操作传播到相关实体。
- CascadeType. REFRESH:把对实体的 refresh 操作传播到相关实体。
- CascadeType. DETACH:把对实体的 detach 操作传播到相关实体。

关系标注(包括@OneToMany、@OneToOne、@ManyToOne 和@ManyToMany 等)都有一个 cascade 属性。该属性的类型是 CascadeType[],所以可以指定多个层叠操作类型值,如 cascade={CascadeType. PERSIST, CascadeType. REMOVE}。

对所有的关系标注,其 cascade 属性的默认值为空,即不会把任何对实体的操作传播到相关实体。通常,对@OneToOne 标注和@OneToMany 标注,其 cascade 属性值可以取 CascadeType. ALL,对@ManyToOne 标注和@ManyToMany 标注,其 cascade 属性值则取默认值。

4. 合并实体实例

EntityManager 的 merge 方法可用于更新一个实体实例及其相关实体实例的持久性数据。merge 方法将一个分离的实体实例的状态纳入到相同身份的已存在或新建的受管理的实体实例中,方法返回受管理的实体实例。

merge 方法的方法签名如下:

```
public <T>T merge(T entity)
```

使用说明:

(1) 调用方法时,必须指定一个实体实例参数。如果指定的参数不是实体类型对象,将抛出 java. lang. IllegalArgumentException 型例外。

(2) 方法调用后,指定的实体实例仍然是分离的,而方法返回的实体实例是受管理的。

(3) 若指定的实体实例是新的,则新建一个受管理的实体实例,并将指定实体实例的状态纳入该新建的受管理的实体实例。方法返回新建的受管理的实体实例。

(4) 若指定的实体实例是删除的,方法抛出一个 IllegalArgumentException 型例外。

(5) 若指定的实体实例本来是受管理的,则该操作被忽略。

若进行合并操作的实体实例(包括受管理的实体实例)存在相关的实体实例,且关系标注(如@OneToMany、@OneToOne 等)的 cascade 属性被设置为 MERGE 或 ALL,则进行层叠合并操作,即相关的实体实例也进行合并操作。

5. 删除实体实例

要从数据库中删除一条记录,可以先获得该记录相应的受管理实体实例,然后再调用 EntityManager 的 remove 方法。remove 方法使一个受管理的实体实例成为删除的,删除的实体实例相应的数据记录会在所在事务递交前或执行 flush 操作时从数据库中删除。

remove 方法的方法签名如下:

```
public void remove(Object entity)
```

使用说明:

(1) 调用方法时,必须指定一个实体实例参数。如果指定的参数不是实体类型对象,将抛出 java.lang.IllegalArgumentException 型例外。

(2) 若指定的实体实例是新的,则该操作被忽略,即该实体实例仍然是新的。

(3) 若指定的实体实例是删除的,则该操作被忽略,即该实体实例仍然是删除的。

(4) 若指定的实体实例是分离的,方法将抛出 IllegalArgumentException 型例外。

若进行删除操作的实体实例是新的或受管理的、并存在相关的实体实例,且关系标注(如@OneToMany、@OneToOne)的 cascade 属性被设置为 REMOVE 或 ALL,则进行层叠持久操作,即相关的实体实例也进行删除操作。

代码清单 12-4 演示了 merge 和 remove 方法的使用。方法接收一个分离 Client 型实体实例,并将其纳入持久性上下文,然后通过 remove 方法从数据库中删除该实体实例相应的客户记录以及与该客户记录相关的标题记录和回复记录。这里,假设 Client 实体类所有关系变量(包括 replyCollection、topicCollection 和 topicCollection1)的@OneToMany 标注的 cascade 属性值都为 CascadeType.ALL 或 CascadeType.REMOVE。

代码清单 12-4　merge 和 remove 方法使用示例

```
public void delete(Client client){
  EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
  EntityManager em=emf.createEntityManager();
  try {
    em.getTransaction().begin();
    Client c=em.merge(client);
    em.remove(c);
    em.getTransaction().commit();
  }catch(Exception p){
  }
  em.close();
```

```
    emf.close();
}
```

6. 分离实体实例

EntityManager 的 detach 方法使一个受管理的实体实例或删除的实体实例脱离持久性上下文，成为分离的实体实例。该方法不会让持久性上下文执行刷新数据库操作，即被分离的实体实例及其状态数据不会同步至数据库。

detach 方法的方法签名如下：

```
public void detach(Object entity)
```

使用说明：

(1) 调用方法时，必须指定一个实体实例参数。如果指定的参数不是实体类型对象，将抛出 java.lang.IllegalArgumentException 型例外。

(2) 若指定的实体实例是新的，则该操作被忽略，即该实体实例仍然是新的。

(3) 若指定的实体实例是分离的，则该操作被忽略，即该实体实例仍然是分离的。

若进行分离操作的实体实例存在相关的实体实例，且关系标注（如@OneToMany 等）的 cascade 属性被设置为 DETACH 或 ALL，则进行层叠持久操作，即相关的实体实例也进行分离操作。

7. 刷新数据库

刷新数据库是指将受管的实体实例状态同步至数据库，或从数据库中移去删除实体实例对应的数据记录。刷新模式有两种，定义在枚举类型 FlushModeType 中：

- FlushModeType.AUTO。
- FlushModeType.COMMIT。

默认情况下，刷新模式为 FlushModeType.AUTO。此时，持久性提供器会根据需要自动执行刷新数据库操作，一般发生在事务提交前或在实体管理器产生的查询对象上执行查询操作前。

可以调用 EntityManager 的 setFlushMode 方法改变刷新模式。比如下面代码将刷新模式设置为 FlushModeType.COMMIT：

```
em.setFlushMode(FlushModeType.COMMIT);
```

当刷新模式为 FlushModeType.COMMIT 时，持久性提供器就只在事务提交前执行刷新数据库操作。

无论是哪种刷新模式，都可以在需要时调用 EntityManager 的 flush 方法显式操作刷新数据库操作。flush 方法的方法签名如下：

```
public void flush()
```

一旦调用此方法，持久性提供器就会把每个受管理的实体实例的状态同步到数据库，也会将删除实体实例对应的数据记录从数据库中移去。

flush 方法只能在事务活动的情况调用，否则将抛出 TransactionRequiredException 型例外。如果不存在活动的事务，持久性提供器不会进行刷新数据库操作，包括在执行查询操作前。

8. 刷新实体实例

刷新实体实例是指用数据库数据重置一个受管理的实体实例的状态。EntityManager 的 refresh 方法可以实现这一操作，该方法的方法签名如下：

```
public void refresh(Object entity)
```

使用说明：

(1) 如果指定的参数对象不是一个实体实例，或不是一个受管理的实体实例，方法将抛出一个 IllegalArgumentException 型例外。

(2) 如果数据库中不存在与指定实体实例有相同身份的实体记录，方法将抛出一个 EntityNotFoundException 型例外。

若进行刷新操作的实体实例存在相关的实体实例，且关系标注(如@OneToMany 等)的 cascade 属性被设置为 REFRESH 或 ALL，则进行层叠持久操作，即相关的实体实例也进行刷新操作。

9. 清空持久性上下文

EntityManager 的 clear 方法使与持久性上下文相关联的实体实例(包括受管理的和删除的)成为分离的实体实例。clear 方法的方法签名如下：

```
public void clear()
```

该方法不会让持久性上下文执行刷新数据库操作，即实体实例的新状态不会同步至数据库，与删除的实体实例同身份的实体记录也不会从数据库移去。

12.3 事务控制

除查找(find)、查询(Query)等访问操作外，其他涉及实体实例新建、更新和删除等的操作一般都应该在事务内执行，以便在事务递交时，能将新建、更新和删除的实体实例同步至数据库。

本地资源型事务的控制在应用程序编程接口 EntityTransaction 中定义。每个本地资源型实体管理器(EntityManager)对象都对应有一个 EntityTransaction 型对象，调用实体管理器对象的 getTransaction 方法可以返回该 EntityTransaction 型对象。EntityTransaction 接口的常用方法如表 12-2 所示。

表 12-2 EntityTransaction 接口的常用方法

方法	描述	方法	描述
Void begin()	开始一个本地资源事务	Void rollback()	回滚当前事务
Void commit()	递交当前事务	boolean isActive()	检测事务是否在进行中

对于一个 EntityTransaction 型对象，可以依次执行多个事务。每个事务由调用 begin 方法开始，到调用 commit 或 rollback 方法结束。在此期间，isActive 方法返回 true，表明事务正在进行中。

如果在调用 begin 方法时事务已在进行中，或者在调用 commit 或 rollback 方法时事务

没有在进行中,都将抛出 IllegalStateException 例外。

12.4 JPQL

Java 持久性查询语言(JPQL)用于定义基于实体的查询,以及批量更新和删除操作。JPQL 与 SQL 的语法非常相似,但 JPQL 查询的主体是实体和实例变量,而非表和列。它使应用程序开发者可以定义独立于底层数据库、可移植的查询语句,当实体映射被改变时,不需要改动 JPQL 查询。

JPQL 支持基于@NamedQuery 标注的静态查询,也支持在运行时构建并处理的动态查询。无论是静态查询还是动态查询,都可以采用参数绑定。

JPQL 语句包括 SELECT 语句、UPDATE 语句和 DELETE 语句。

12.4.1 SELECT 语句格式

SELECT 语句由 SELECT 子句、FROM 子句、WHERE 子句、GROUP BY 子句、HAVING 子句和 ORDER BY 子句组成,其中 SELECT 子句和 FROM 子句是必需的,其他子句是可选的。

```
<SELECT 子句><FROM 子句>[<WHERE 子句>]
[<GROUP BY 子句>] [<HAVING 子句>] [<ORDER BY 子句>]
```

说明:在 JPQL 语法格式中,符号<>表示该项应根据需要和情况具体指定;符号[]表示该项为可选项。

12.4.2 标识变量

标识变量表示某种实体类型的一个实例。标识变量是在 FROM 子句中声明的一个标识符。标识变量只能在 FROM 子句中声明,不能在其他子句中声明,但通常会在其他子句中使用。

标识变量不能与 SELECT、UPDATE、DELETE、AS、FROM 等 JPQL 的保留字同名,也不能是同一持久性单元中某个实体类型名。在 JPQL 中,标识变量和保留字都是不区分大小写的。

有三种格式可以声明标识变量:范围变量声明、JOIN 短语以及集合成员声明。

1. 范围变量声明

范围变量声明定义一个表示指定实体类型实体的标识变量。范围变量声明的语法格式如下:

```
<实体类型> [AS] <标识变量>
```

下面语句可以查询所有的客户,其中 FROM 子句中声明了一个名为 c 的标识变量,表示一个客户实体。

```
SELECT c FROM Client AS c
```

或

```
SELECT c FROM Client c
```

2. JOIN 短语

JOIN 短语总是紧跟在范围变量声明或其他 JOIN 短语之后(相互之间用空格分隔),用于声明一个表示与之前已定义的标识变量相关的实体的标识变量。JOIN 短语的语法格式如下:

```
[LEFT [OUTER]|INNER] JOIN
{<标识变量>.<单值关系变量>|<标识变量>.<集合值关系变量>}
[AS] <标识变量>
```

这里,JOIN 或 INNER JOIN 表示内连接,LEFT JOIN 或 LEFT OUTER JOIN 表示左连接。与 SQL 中的 JOIN 短语相比较,JPQL 中的 JOIN 短语隐含有连接条件,即查询的是相关的实体,而不会把无关的实体连接在一起。

说明:在 JPQL 语法格式中,符号|可以将两项或多项连接起来,表示选择其中一项。为标明第一项的开始处及最后一项的结尾处,可用符号{}将这些选项括起来。

下面语句查询所有发布过主题的客户。FROM 子句中声明了两个标识变量:c 和 t。c 表示一个客户实体,t 表示一个相关的主题实体。这里,表达式 c.topicCollection 引用了之前已经声明的标识变量 c,topicCollection 是 Client 实体类中定义的一个集合值关系变量,点是导航运算符,表示从一个客户导航至与其相关的一组主题实体。

```
SELECT DISTINCT c FROM Client c JOIN c.topicCollection t
```

说明:上述 JPQL SELECT 语句的功能与下面的 SQL SELECT 语句相当:

```
SELECT DISTINCT c.*
FROM TOPIC t,CLIENT c WHERE(t.username=c.username)
```

上述语句的查询功能也可以用下面语句等价实现。

```
SELECT c FROM Client c WHERE c.topicCollection IS NOT EMPTY
```

说明:上述 JPQL SELECT 语句的功能与下面的 SQL SELECT 语句相当:

```
SELECT *
FROM CLIENT c
WHERE ((SELECT COUNT(t.ID) FROM TOPIC t WHERE (t.USERNAME=c.USERNAME))>0)
```

3. 集合成员声明

集合成员声明定义一个表示某实体集合中一个成员的标识变量。集合成员声明的语法格式如下,其中的集合值路径表达式表示某实体集合:

```
IN(<集合值路径表达式>) [AS] <标识变量>
```

下面语句实现与上面“JOIN 短语”中介绍的语句相同的功能,即查询所有发布过主题的客户。

```
SELECT DISTINCT c FROM Client c, IN (c.topicCollection) t
```

12.4.3 路径表达式

路径表达式在前面 JOIN 短语和集合成员声明中已经出现过(c.topicCollection),它可以通过实体类中的关系变量定义实体之间的导航路径。在 JPQL 中,路径表达式是一个重要的语法成分,它可能出现在 SELECT 语句的各个子句中,也会用于 UPDATE 和 DELETE 语句中。

路径表达式分为单值路径表达式和集合值路径表达式,单值路径表达式又分为单值关系路径表达式和状态变量路径表达式。

1. 单值关系路径表达式

单值关系路径表达式的语法格式如下:

```
<标识变量>.{<单值关系变量>.}*<单值关系变量>
```

说明:在 JPQL 语法格式中,符号{}*表示该项可重复 0 至多次。

下面语句查询由用户名为"zhaoyy"的客户发布的主题的所有回复。这里,WHERE 子句中的 r.topic.client 是一个单值关系路径表达式,其中 r 是一个 Reply 类型的标识变量,topic 是 Reply 实体类中定义的一个 Topic 类型的单值关系变量,client 是 Topic 实体类中定义的一个 Client 类型的单值关系变量。

```
SELECT r FROM Reply r,Client c WHERE r.topic.client=c AND c.username='zhaoyy'
```

说明:上述 JPQL SELECT 语句的功能与下面的 SQL SELECT 语句相当:

```
SELECT r.*
FROM CLIENT c,REPLY r,TOPIC t
WHERE ((c.USERNAME='zhaoyy') AND ((t.ID=r.TOPID) AND (c.USERNAME=t.USERNAME)))
```

2. 状态变量路径表达式

状态变量路径表达式的格式如下:

```
{<标识变量>.<状态变量>|<单值关系路径表达式>.<状态变量>}
```

下面语句实现与上面"单值关系路径表达式"中介绍的语句相同的功能,即查询由用户名为"zhaoyy"的客户发布的主题的所有回复。这里,r.topic.client.username 是一个状态变量路径表达式,其中 r 是标识变量,topic 和 client 都是单值关系变量,username 是 Client 实体类中定义的持久性状态变量。

```
SELECT r FROM Reply r WHERE r.topic.client.username='zhaoyy'
```

3. 集合值(关系)路径表达式

集合值(关系)路径表达式的格式如下:

```
<标识变量>.{<单值关系变量>.}*<集合值关系变量>
```

下面语句查询既没有发表主题也没有对任何主题作过回复的客户。这里,WHERE 子句中的 c.topicCollection 和 c.replyCollection 都是集合值路径表达式。其中 c 是一个 Client 类型的标识变量,topicCollection 和 replyCollection 都是 Client 实体类中定义的集合

值关系变量。

```
SELECT c FROM CLIENT c WHERE c.topicCollection IS EMPTY AND c.replyCollection IS EMPTY
```

说明：上述 JPQL SELECT 语句的功能与下面的 SQL SELECT 语句相当：

```
SELECT c.*
FROM CLIENT c
WHERE (
    ((SELECT COUNT(t.ID) FROM TOPIC t WHERE (t.USERNAME=c.USERNAME))=0)
    AND
    ((SELECT COUNT(r.ID) FROM REPLY r WHERE (r.USERNAME=c.USERNAME))=0))
```

从上面的语法格式可以看出，在路径表达式中，由单值关系变量可以导航至状态变量、其他的单值关系变量或者集合值关系变量，但由状态变量或集合值关系变量则无法导航至其他的元素。

每个路径表达式都有确切的 Java 类型。路径表达式的类型由路径表达式中最后一个元素的类型确定，如状态变量路径表达式 r.topic.client.username 的类型是状态变量 username 的类型，集合值路径表达式 c.topicCollection 的类型是集合值关系变量 topicCollection 的类型。

12.4.4 FROM 子句

FROM 子句指定查询的范围，其语法格式如下：

```
FROM <范围变量声明>{JOIN 短语}*{,{<范围变量声明>{JOIN 短语}*|<集合成员声明>}}*
```

FROM 子句由一些标识变量声明组成：首先是范围变量声明，然后可以跟 0 至多个范围变量声明或集合成员声明，两者之间用逗号分隔。每个范围变量声明后可跟 0 至多个 JOIN 短语，JOIN 短语可看作是范围变量声明的附属。

12.4.5 SELECT 子句

SELECT 子句指定查询结果的类型，其语法格式如下：

```
SELECT [DISTINCT] <SELECT_表达式>{,<SELECT_表达式>}*
```

其中，可选项 DISTINCT 表示从查询结果中消除重复项，SELECT_表达式的类型确定了查询返回的那些对象或值的类型。SELECT 子句可以包含 1 个或多个 SELECT_表达式。如果仅包含一个 SELECT_表达式，查询返回的对象或值的类型即为该表达式的类型。如果包含多个 SELECT_表达式，查询返回的对象或值的类型为 Object[]，其中各元素的次序和具体类型与子句中列出的各表达式对应。

SELECT_表达式可以是以下语法成分：

- 标识变量。
- 单值路径表达式。
- 聚合表达式。
- 构造方法表达式。

使用聚合函数可以先对查询的初始结果进行计算，然后再返回最终的查询结果。聚合表达式的语法格式如下：

```
{AVG|MAX|MIN|SUM}([DISTINCT]<状态变量路径表达式>)|
COUNT([DISTINCT] {<标识变量>|<状态变量路径表达式>|<单值关系路径表达式>})
```

表 12-3 列出了用于聚合表达式中的聚合函数的特性。

表 12-3　聚合函数的特性

名　　称	功　　能	返 回 类 型
AVG	计算平均值	Double
COUNT	计数	Long
MAX	计算最大值	其应用的状态变量路径表达式的类型
MIN	计算最小值	其应用的状态变量路径表达式的类型
SUM	计算总和	Long、Double、BigInteger 或 BigDecimal

使用聚合函数进行查询和计算时，如果没有查询结果可用于计算，那么对于 AVG、MAX、MIN 和 SUM 函数，计算结果为 null，对于 COUNT 函数，计算结果为 0。

下面语句查询在线(status 值为 1)用户的人数，查询结果是一个 Long 型实例对象。

```
SELECT COUNT(c) FROM Client c WHERE c.status='1'
```

构造方法表达式的格式如下：

```
NEW <构造方法名>(<构造项>{,<构造项>}*)
```

其中，构造方法名也即类名，构造项可以是单值路径表达式或聚合表达式。使用构造方法表达式，可以使查询返回一个或多个指定类的实例对象，而不是一个或多个 Objecc[]型对象，从而更便于后续处理。指定的类不必映射到数据库，也不必是实体类。

12.4.6　WHERE 子句

WHERE 子句指定查询、更新或删除的条件，其基本的语法格式如下：

```
WHERE <条件表达式>
```

条件表达式由文字、标识变量、路径表达式、输入参数与各种运算符连接组成，其类型应该是布尔型。表 12-4 列出了 JPQL 支持的各种运算符及其优先级。

表 12-4　JPQL 运算符及其优先级

分　　类	运　算　符	优先级
导航运算符	.	1
算术运算符	+、-(单目)	2
	*、/(乘、除)	3
	+、-(加、减)	4

续表

分　类	运　算　符	优先级
关系运算符	=、>、>=、<、<=、<>	5
	[NOT] BETWEEN、[NOT] LIKE	
	[NOT] IN、IS [NOT] NULL、IS [NOT] EMPTY、[NOT] MEMBER [OF]	
逻辑运算符	NOT	6
	AND	7
	OR	8

1. BETWEEN 表达式

使用 BETWEEN 表达式可以测试某表达式的值是否落在指定的区间，其语法格式如下：

```
<测试表达式> [NOT] BETWEEN <起始值表达式>AND <终止值表达式>
```

其中，三个表达式的类型必须相同，可以是数值型，也可以是字符串或日期时间型。

下面两个 BETWEEN 表达式的功能相当：

```
t.clickcount BETWEEN 100 AND 200
t.clickcount>=100 AND t.clickcount<=200
```

下面两个表达式也具有相同的功能：

```
t.clickcount NOT BETWEEN 100 AND 200
t.clickcount<100 OR t.clickcount>200
```

2. IN 表达式

使用 IN 表达式可以测试一个状态变量路径表达式的值是否属于某个值列表或子查询，其语法格式如下：

```
<状态变量路径表达式> [NOT] IN({<in_项>{,<in_项>} * |<子查询>})
```

其中 in_项可以是以下语法成分：

- 文字。
- 输入参数。

下面表达式用以测试一个客户的 username 值是否为“liling”、“zhaoyy”或“lilong”。

```
c.username IN ('liling', 'zhaoyy', 'lilong')
```

3. LIKE 表达式

使用 LIKE 表达式可以进行字符串的匹配查询，其语法格式如下：

```
<字符串表达式> [NOT] LIKE {<字符串文字> |<字符串输入参数>}
```

在字符串文字或字符串输入参数中，下划线(_)代表任意单个字符，百分号（%)代表任意的字符序列(包括空)。

下面表达式测试客户的邮箱地址是否以“163.com”结尾。

```
c.email LIKE '%163.com'
```

4. NULL 比较表达式

用于测试一个单值路径表达式或输入参数的值是否为 NULL 值，其语法格式如下：

```
{<单值路径表达式>|<输入参数>} IS [NOT] NULL
```

5. 空集合比较表达式：

用于测试一个集合值路径表达式的值是否包含元素，其语法格式如下：

```
<集合值路径表达式>IS [NOT] EMPTY
```

如果集合值路径表达式的值本身为 NULL 值，则整个表达式的值也为 NULL。

下面语句查询所有已有回复的主题。

```
SELECT t FROM Topic t WHERE t.replyCollection IS NOT EMPTY
```

6. 输入参数

在 WHERE 和 HAVING 子句中，可以引入输入参数，以便增加查询的通用性和灵活性。JPQL 支持两种类型的输入参数：命名参数和位置参数。

- 命名参数：以冒号(:)开头、后跟一个字符串，如：name，其中字符串作为输入参数的名称是大小写敏感的。
- 位置参数：以问号(?)开头、后跟一个整数，如：?1，其中整数作为输入参数的编号必须大于等于 1。

在同一个查询中，可以引入多个输入参数，但不应混合使用命名参数和位置参数。不同的参数应该有不同的名称或编号，但同一个参数可以出现多次。

关于参数的设置及 JPQL 语句的执行请参见第 12.5 节。

7. 集合成员比较表达式

用于测试一个值是否为一个集合的成员，其中被测试的值和集合成员必须有相同的数据类型。其语法格式如下：

```
<实体表达式>[NOT] MEMBER [OF] <集合值路径表达式>
```

这里，实体表达式可以是单值关系路径表达式、标识变量或输入参数。

下面语句返回包含指定回复的主题。

```
SELECT t FROM Topic t WHERE :reply MEMBER t.replyCollection
```

8. EXISTS 表达式

使用 EXISTS 表达式可以测试一个子查询是否存在查询结果，其语法格式如下：

```
[NOT] EXISTS(<子查询>)
```

下面语句查询所有没有任何回复的主题。

```
SELECT t FROM Topic t WHERE NOT EXISTS (SELECT r FROM Reply r WHERE r.topic=t)
```

该查询的查询主体是主题。对每一个主题，主查询向子查询提供该主题，子查询查找所

有属于该主题的回复，然后主查询判断子查询是否存在查询结果，若不存在，则当前主题是需要查询的。

9. ALL 或 ANY 表达式

该表达式用于比较主查询中的某个表达式值与子查询的查询结果，其语法格式如下：

```
<表达式><比较运算符>{ALL|ANY|SOME}(<子查询>)
```

这里的比较运算符包括：＝、＜＞、＞、＞＝、＜和＜＝。如果选用 ALL，那么只有主查询中表达式值与子查询的所有查询结果都符合比较要求，整个表达式的值才为 TRUE，否则整个表达式的值为 FALSE。如果选用 ANY，那么主查询中表达式值只要与子查询的某个查询结果符合比较要求，整个表达式的值就为 TRUE，只有当主查询中表达式值与子查询的所有查询结果都不符合比较要求，整个表达式的值才为 FALSE。关键字 ANY 与 SOME 是同义词，可以互相替换。

下面语句查询最新发布的主题，即发布这些主题后还没有任何客户对任何主题发表过回复。

```
SELECT t FROM Topic t WHERE t.createtime >ALL (SELECT r.replytime FROM Reply r)
```

子查询从回复表中获取所有回复的发表时间，主查询从主题表中查询所有创建时间大于子查询获得的所有时间的主题。

12.4.7 GROUP BY 和 HAVING 子句

使用 GROUP BY 子句可以对实体进行分组汇总查询，该子句的语法格式如下：

```
GROUP BY <单值路径表达式>{,<单值路径表达式>}*
```

其中子句中指定的单值路径表达式为分组标识，即在这些表达式上取值相同的实体为一组。分组汇总查询通常会利用聚合函数对每一组实体进行计算。

下面语句查询各类文化程度（不同 degree 值）的客户人数。

```
SELECT c.degree, COUNT(c) FROM Client c GROUP BY c.degree
```

HAVING 子句总是与 GROUP BY 子句配合使用，用于指定分组查询的条件，即哪些分组汇总后满足查询条件。该子句的语法格式如下：

```
HAVING <条件表达式>
```

当 SELECT 语句中既包含 WHERE 子句，又包含 GROUP BY 和 HAVING 子句时，首先用 WHERE 子句过滤查询内容，然后再基于 GROUP BY 子句对过滤后的内容进行聚合计算，最后用 HAVING 子句过滤聚合后的内容并返回最终的查询结果。

12.4.8 ORDER BY 子句

ORDER BY 子句用于对查询结果按指定的状态变量路径表达式的值进行排序，该子句的语法格式如下：

```
ORDER BY <状态变量路径表达式>[ASC|DESC] {,<状态变量路径表达式>[ASC|DESC]}*
```

如果指定了多个状态变量路径表达式，那么先按前面的状态变量路径表达式值进行排序，若其值相同，再按后面的状态变量路径表达式值进行排序。ASC 表示升序，DESC 表示降序，其中 ASC 是默认值。

下面语句查询所有主题，并按创建时间对主题进行降序排序，即后发布的主题在前，先发布的主题在后。

```
SELECT t FROM Topic t ORDER BY t.createtime DESC
```

12.4.9 UPDATE 和 DELETE 语句

UPDATE 语句可以实现对实体的批量更新，其基本语法格式如下：

```
UPDATE <实体类型>[AS] <标识变量>
SET [<标识变量>.]{<状态变量>|<单值关系变量>}=<新值>
    {,[<标识变量>.]{<状态变量>|<单值关系变量>}=<新值>}*
[WHERE <条件表达式>]
```

其中，UPDATE 子句指定要更新的实体类型，SET 短语指定更新的具体内容，WHERE 子句指定更新条件，即对哪些实体进行更新。WHERE 子句的书写规则与 SELECT 语句中的相同。如果缺省 WHERE 子句，将对指定类型的所有实体进行更新。

下面语句对所有由“zhaoyy”发布的主题的 clickcount（点击数）加 1。

```
UPDATE Topic t SET t.clickcount=t.clickcount+1 WHERE t.client.username='zhaoyy'
```

这里，SET 短语中赋值号左边状态变量或单值关系变量前的标识变量是可省略的，所以上面语句也可以写成如下。

```
UPDATE Topic t SET clickcount=t.clickcount+1 WHERE t.client.username='zhaoyy'
```

DELETE 语句可以实现对实体的批量删除，其语法格式如下：

```
DELETE FROM <实体类型>[AS] <表示变量>[WHERE <条件表达式>]
```

其中，DELETE 子句指定要删除的实体类型，WHERE 子句指定删除条件，即要删除哪些实体。如果缺省 WHERE 子句，将删除指定类型的所有实体。

12.5 执行 JPQL 语句

JPQL 虽然在语法上与 SQL 相似，但并不能在数据库系统中直接执行。JPQL 语句的执行首先需要被转换成持久性提供器当前相连接的数据库系统的 SQL 语句，然后再把 SQL 语句执行后返回的数据装配成实体实例或其他 Java 对象返回。这个过程需要持久性提供器支持，并通过查询 API 实现，如图 12-4 所示。

12.5.1 基本过程

在 JPA 中，执行 JPQL 语句的基本步骤如下：

（1）获得实体管理器对象；

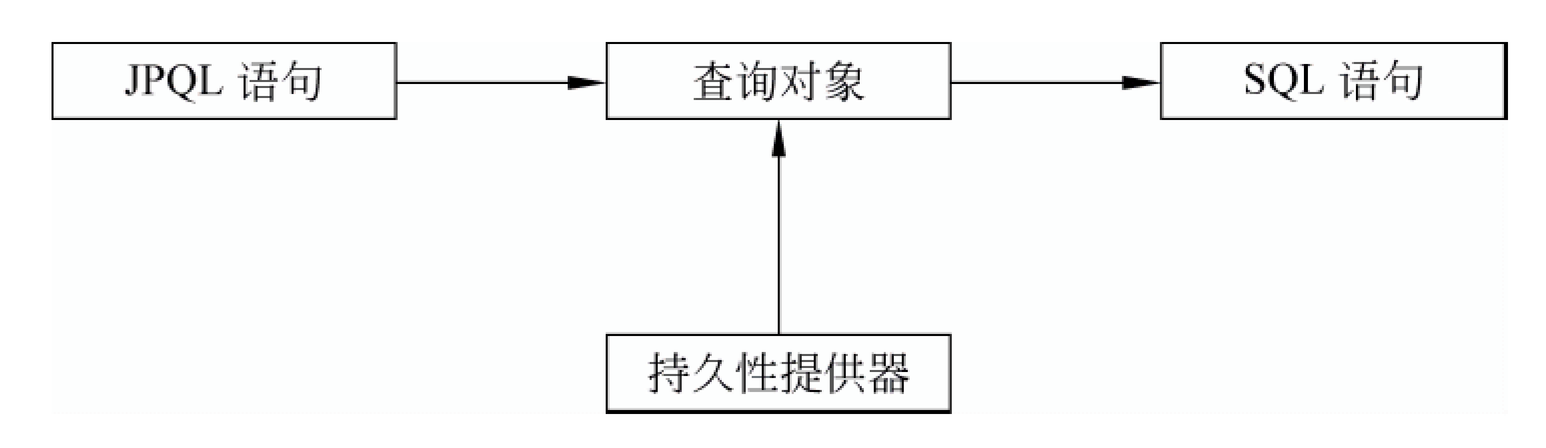


图 12-4 JPQL 语句的执行

(2) 创建包含 JPQL 语句的查询对象;

(3) 调用查询对象的相关方法执行查询;

(4) 浏览或处理查询结果(实体)。

要利用 JPQL 进行查询,首先要定义 JPQL 语句。这里有两种定义 JPQL 语句的方式。一是所谓的动态查询,即在程序代码中创建 JPQL 语句字符串;二是所谓静态查询或命名查询,是指用@NamedQuery 标注定义 JPQL 语句并对其命名。下面是@NamedQuery 标注的一个示例,其中 query 属性指定了 JPQL 语句,name 属性指定了名称。

```
@NamedQuery(name="Client.findAll",query="SELECT c FROM Client c")
```

@NamedQuery 标注应该修饰于实体类。因为一个程序元素只能有某种标注类型的一个标注,所以一个实体类也只能有一个@NamedQuery 标注。如果需要在一个实体类中定义多个命名查询,可以借助@NamedQueries 标注。@NamedQueries 标注仅有一个 value 属性,该属性的类型是@NamedQuery[],所以一个@NamedQueries 标注可以包含多个@NamedQuery 标注。下面是@NamedQueries 标注的一个示例,其中定义了三个命名查询。

```
@NamedQueries({
    @NamedQuery(name="Client.findAll",query="SELECT c FROM Client c"),
    @NamedQuery(name="Client.findByUsername",
        query="SELECT c FROM Client c WHERE c.username=:username"),
  @NamedQuery(name="Client.findByStatus",
        query="SELECT c FROM Client c WHERE c.status=:status")
})
```

需要注意的是,在一个持久性单元范围内,所有命名查询的名称必须是唯一的。

第 11 章介绍的通过数据库自动生成的实体类中,类的标注中都会包含有关该实体的命名查询,查询的名称就如上述例子一样,都由相应的实体类名来限制。这样可以确保持久性单元内各命名查询的名称是互不相同的。

定义好了 JPQL 语句,就可以按上述步骤执行 JPQL 语句了。首先要获得实体管理器对象,然后利用实体管理器创建包含 JPQL 语句的查询对象。

下面是一些用于创建查询对象的 EntityManager 方法。

- `public Query createQuery(String qlString);`

  创建一个查询对象,用于执行指定的 JPQL 语句。

- `public <T>TypedQuery<T>createQuery(String qlString,Class<T>resultClass);`

创建一个类型化的查询对象,用于执行指定的 JPQL 语句。该 JPQL 语句的查询结果只能指定单一项,方法的第 2 个参数指定该单一项类型。

- public Query createNamedQuery(String name);

  创建一个查询对象,用于执行指定的 JPQL 命名查询。

- public <T>TypedQuery<T>createNamedQuery(String name,Class<T>resultClass);

  创建一个类型化的查询对象,用于执行指定的 JPQL 命名查询。该命名查询的查询结果只能指定单一项,方法的第 2 个参数指定该单一项类型。

下面代码共创建了四个查询实例:两个是动态查询实例 query1 和 query3,另外两个是命名查询实例 query2 和 query4。在这里,动态查询中包含了一个位置参数?1,命名查询中包含了一个命名参数:status。

```
EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
EntityManager em=emf.createEntityManager();
Query query1=em.createQuery("SELECT c FROM Client c WHERE c.useranme=?1");
Query query2=em.createNamedQuery("Client.findByStatus");
TypedQuery<Client>query3=em.createQuery(
    "SELECT c FROM Client c WHERE c.useranme=?1",Client.class);
TypedQuery<Client>query4=em.createNamedQuery("Client.findByStatus",Client.class);
……
em.close();
emf.close();
```

说明:命名查询 Client.findByStatus 的定义可查看通过数据库自动生成的 Client 实体类中的相关标注。

当然,无论是动态查询还是命名查询,都可以使用位置参数和命名参数,但要保证一个查询中不能混合使用两种参数。

另外,query1 和 query2 是两个未类型化的查询对象,而 query3 和 query4 是两个类型化的查询对象。通过 query3 或 query4 查询获得的单个结果或集合中的元素都具有 Client 类型。

12.5.2 查询 API

有了查询对象,接下来就可以通过调用查询对象的相关方法执行它所包含的 JPQL 语句,获取查询结果。表 12-5 列出 Query 对象的一些常用方法。TypedQuery 对象具有的方法与此类似,这里不再列出。

表 12-5 Query 接口的常用方法

方　　法	描　　述
Object getSingleResult()	检索单一实体或对象
List getResultList()	检索实体集合
int executeUpdate()	执行一个更新或删除语句

续表

方　　法	描　　述
Query setParameter(String name, Object value)	设置命名参数的值
Query setParameter(int position, Object value)	设置位置参数的值
Query setMaxResults(int maxResult)	设置要检索对象的最大数量
Query setFirstResult(int startPosition)	设置第一个结果的位置(从 0 开始)

1. 设置查询参数

如果在查询对象时指定的 JPQL 语句中包含输入参数,那么在执行查询前先要设置相关参数。setParameter(int, Object)方法用于设置位置参数,其中第一个参数指定输入参数的编号,第二个参数指定输入参数的值;setParameter(String, Object)方法用于设置命名参数,其中第一个参数指定输入参数的名称,第二个参数指定输入参数的值。

下面代码演示如何设置位置参数和命名参数。

```
query1.setParameter(1, "zhaoyy");
query2.setParameter("status", '1');                    //用户状态为在线
query3.setParameter(1, "zhaoyy");
query4.setParameter("status", '1');
```

这些设置查询参数的方法都返回查询实例本身,所以通常可以省略返回值。

2. 查询单一实体

查询对象的 getSingleResult 方法用于查询单一实体(或对象),此时应确保查询的结果有一个实体、且仅有一个实体。如果不存在任何实体,方法抛出 NoResultException 型例外。如果存在多个实体,方法抛出 NonUniqueResultException 型例外。

对于 Query 对象,该方法的返回类型是 Object,可以强制转换成确切的实体类型。对于 TypedQuery 对象,该方法的返回类型在创建查询对象时就已确定。下面代码查询已经指定了用户名的一个客户实体。

```
Client client1=(Client)query1.getSingleResult();
Client client3=query3.getSingleResult();
```

3. 查询实体集合

查询对象的 getResultList 方法用于查询实体(或对象)集合,方法返回一个 List 表。对于 Query 对象,该方法返回的 List 表的元素类型是未知的,而对于 TypedQuery 对象,该方法返回的 List 表的元素类型是确定的。

下面代码查询所有在线客户实体。对前一条代码,编译器无法确定返回的 List 表的元素类型是否为 Client,所以会出现编译警告信息,此时程序员应确保返回的 List 表的元素类型是 Client。对后一行代码,由于 query4 是类型化的查询对象,所以编译器可以确定返回的 List 表的元素类型是否为 Client,如果不是,将导致编译出错。

```
List<Client>clients2=query2.getResultList();
List<Client>clients4=query4.getResultList();
```

如果不存在任何查询结果，方法返回 null，不会抛出例外。

4. 执行更新或删除

执行 UPDATE 和 DELETE 语句的过程与执行 SELECT 语句的类似，首先要创建相应的查询对象，然后再调用查询对象的 executeUpdate 方法。方法返回被更新或删除的实体数量。

下面代码将指定客户实体的 email 属性值更改为“zhaoyy@gmail.com”。

```
EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
EntityManager em=emf.createEntityManager();
try {
  em.getTransaction().begin();
  Query query=em.createQuery(
    "UPDATE Client c SET c.email='zhaoyy@gmail.com' WHERE c.username='zhaoyy'");
  query.executeUpdate();
  em.getTransaction().commit();
} catch(PersistenceException e){}
em.close();
emf.close();
```

通常，SELECT 语句可以不在事务内执行，此时对返回结果所做的任何修改都不会同步至数据库。而 UPDATE 和 DELETE 语句则必须在事务内执行，否则 executeUpdate 方法将抛出 TransactionRequiredException 例外。

12.6 论坛—重写业务方法

本节继续 11.5 节介绍的应用项目(luntan_final)，为论坛应用重写所有的业务方法，使其真正从已经创建的 luntan 数据库中访问数据。这里，不再创建新的项目，而是在原来应用项目中直接进行。

12.6.1 为论坛应用定义持久性单元

要在应用中通过 EntityManager 方法或查询 API 访问数据库，首先需要定义持久性单元。这里，基于应用项目中原先创建的 JDBC 资源 jdbc/luntan(数据源)，定义一个持久性单元。持久性单元的属性如下。

- 持久性单元名称：luntanPU。
- 持久性提供器：缺省的 EclipseLink(JPA 2.0)。
- 使用 Java 事务 API：否。
- 表生成策略：无。

定义完持久性单元后，相关的定义内容被保存在 persistence.xml 文件中。文件位于项目中“配置文件”节点下。

12.6.2 更改命名查询

通过数据库自动生成的 Client、Topic 和 Reply 实体类中，都会包含一组相关的命名查

询定义。根据处理需要，这里对其略作改动。

首先修改 Topic 实体类中的 Topic.findAll 命名查询定义，将其 query 属性值由：

```
SELECT t FROM Topic t
```

改成

```
SELECT t FROM Topic t ORDER BY t.lastreplytime DESC
```

使查询返回的主题按其最后回复时间降序排序。

然后为 Reply 实体类增加一个命名查询，查询的名称为 Reply.findByTid，其 query 属性值为

```
SELECT r FROM Reply r JOIN r.topic AS t WHERE t.id=:id ORDER BY r.replytime
```

使查询返回的指定主题编号的所有回复按各回复的回复时间升序排序。

12.6.3 重写业务方法

这里需要对 model.ClientManager、model.TopicManager 和 model.ReplyManager 类中的所有方法进行重新定义。

代码清单 12-5 是 ClientManager 类重写后的完整代码，除对原有方法进行重新定义，还添加了一个新的 init()方法。init 可以将数据库中所有客户的状态设置为 0(不在线)。

代码清单 12-6 是 TopicManager 类重写后的完整代码。代码清单 12-7 是 ReplyManager 类重写后的完整代码。

代码清单 12-5 model.ClientManager 类

```
package model;
import entity.Client;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.PersistenceException;
import javax.persistence.Query;

public class ClientManager {
public Client findClientByName(String name){      //根据用户名获取客户
    EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
    EntityManager em=emf.createEntityManager();
    Client client=em.find(Client.class,name);
    em.close();
    emf.close();
    return client;
  }
public boolean insertClient(Client client){       //插入一个客户
    EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
    EntityManager em=emf.createEntityManager();
```

```
    boolean success=true;
    try {
      em.getTransaction().begin();
      em.persist(client);
      em.getTransaction().commit();
    } catch(PersistenceException e){
      success=false;
    }
    em.close();
    emf.close();
    return success;
  }
  public int getNumOfClient(){                        //获得所有用户数
    EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
    EntityManager em=emf.createEntityManager();
    String sql="SELECT count(c) from Client c";
    Query query1=em.createQuery(sql);
    long n=(Long)query1.getSingleResult();
    em.close();
    emf.close();
    return (int)n;
  }
  public int getNumOfOnline(){                        //获得状态为'1'的用户数
    EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
    EntityManager em=emf.createEntityManager();
    String sql="SELECT count(c) from Client c WHERE c.status='1'";
    Query query1=em.createQuery(sql);
    long n=(Long)query1.getSingleResult();
    em.close();
    emf.close();
    return (int)n;
  }
  public void logoff(Client client){                  //将用户状态设置为'0'
    EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
    EntityManager em=emf.createEntityManager();
    client.setStatus('0');
    try {
      em.getTransaction().begin();
      em.merge(client);
      em.getTransaction().commit();
    } catch(PersistenceException e){
    }
    em.close();
    emf.close();
  }
```

```
    public void login(Client client){                   //将用户状态设置为'1'
      EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
      EntityManager em=emf.createEntityManager();
      client.setStatus('1');
      try {
        em.getTransaction().begin();
        em.merge(client);
        em.getTransaction().commit();
      } catch(PersistenceException e){
      }
      em.close();
      emf.close();
    }
    public void init(){                                 //将所有用户的状态设置为'0'
      EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
      EntityManager em=emf.createEntityManager();
      String sql="UPDATE Client c SET c.status='0'";
      try {
        em.getTransaction().begin();
        Query query=em.createQuery(sql);
        int n=query.executeUpdate();
        em.getTransaction().commit();
      } catch(PersistenceException e){
      }
      em.close();
      emf.close();
    }
}
```

代码清单 12-6　model.TopicManager 类

```
package model;
import entity.Reply;
import entity.Topic;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.PersistenceException;
import javax.persistence.Query;

public class TopicManager {
    public int insertTopic(Topic topic){                //插入一个主题
      EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
      EntityManager em=emf.createEntityManager();
      boolean success=true;
```

```
    try {
      em.getTransaction().begin();
      em.persist(topic);
      em.getTransaction().commit();
    } catch(PersistenceException e){
      success=false;
    }
    em.close();
    emf.close();
    return topic.getId();
  }
  public Topic getTopicById(int id){                //根据 id 返回相应的主题
    EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
    EntityManager em=emf.createEntityManager();
    Topic topic=em.find(Topic.class,id);
    return topic;
  }
  public List<Topic>getTopics(){                    //返回所有的主题
    EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
    EntityManager em=emf.createEntityManager();
    Query query=em.createNamedQuery("Topic.findAll");
    List<Topic>list=query.getResultList();
    return list;
  }
  public void replyTopic(Topic topic,Reply reply){      //主题新增一个回复
    EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
    EntityManager em=emf.createEntityManager();
    topic.setClient1(reply.getClient());
    topic.setLastreplytime(reply.getReplytime());
    topic.setReplycount(topic.getReplycount()+1);
    try {
      em.getTransaction().begin();
      em.merge(topic);
      em.getTransaction().commit();
    } catch(PersistenceException e){
    }
    em.close();
    emf.close();
  }
  public void clickTopic(Topic topic){              //主题的点击数增 1
    EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
    EntityManager em=emf.createEntityManager();
    topic.setClickcount(topic.getClickcount()+1);
    try {
      em.getTransaction().begin();
```

```
      em.merge(topic);
      em.getTransaction().commit();
    } catch(PersistenceException e){
    }
    em.close();
    emf.close();
  }
}
```

代码清单 12-7　model. ReplyManager 类

```
package model;
import entity.Reply;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.PersistenceException;
import javax.persistence.Query;

public class ReplyManager {
  public int insertReply(Reply reply){            //插入一个回复
    EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
    EntityManager em=emf.createEntityManager();
    boolean success=true;
    try {
      em.getTransaction().begin();
      em.persist(reply);
      em.getTransaction().commit();
    } catch(PersistenceException e){
      success=false;
    }
    em.close();
    emf.close();
    return reply.getId();
  }
  public List<Reply>getReplys(int id){     //根据主题 id 返回该主题所有的回复
    EntityManagerFactory emf=Persistence.createEntityManagerFactory("luntanPU");
    EntityManager em=emf.createEntityManager();
    Query query=em.createNamedQuery("Reply.findByTid");
    query.setParameter("id",id);
    List<Reply>list=query.getResultList();
    return list;
  }
}
```

12.6.4 定义和注册系统事件监听器

前面已提及，model. ClientManager 类中新添加了一个 init()方法，可以将数据库中所有注册客户的状态设置为 0(不在线)。但这个方法不会无缘无故被执行，为了使该方法能在应用部署和运行时被调用，需要定义和注册相应的系统事件监听器类。

首先创建一个名为 listener 的 Java 包，然后在包中定义一个系统事件监听器类，可以监听 Application 对象引发的事件，见代码清单 12-8。

代码清单 12-8　系统事件监听器类(listener. LuntanSystemEventListener)

```
package listener;
import javax.faces.application.Application;
import javax.faces.event.SystemEvent;
import javax.faces.event.SystemEventListener;
import model.ClientManager;

public class LuntanSystemEventListener implements SystemEventListener{
  @Override
  public void processEvent(SystemEvent event){
    ClientManager cm=new ClientManager();
    cm.init();
  }
  @Override
  public boolean isListenerForSource(Object source){
    return source instanceof Application;
  }
}
```

然后再创建 Faces 配置文件(faces-config. xml)，并在其中注册上述监听器类，使其可以监听 PostConstructApplicationEvent 型系统事件，见代码清单 12-9。

代码清单 12-9　注册监听器类

```
<application>
  <system-event-listener>
    <system-event-class>
      javax.faces.event.PostConstructApplicationEvent
    </system-event-class>
    <system-event-listener-class>
      listener.LuntanSystemEventListener
    </system-event-listener-class>
  </system-event-listener>
</application>
```

至此，对应用项目 luntan_final 业务方法的重写工作已全部结束。原来的模拟业务数据的 mode. DataBase 类已没有存在的意义了，可以从项目中删除掉。

12.7 小　　结

- 要在应用程序中通过操作实体实现数据库访问，首先需要定义持久性单元。持久性单元定义在 persistence.xml 文件中，该文件存放在 META-INF 目录下。
- 实体的状态包括四种：新的实体实例、受管理的实体实例、分离的实体实例、删除的实体实例。
- 通过 EntityManager API，可以对实体进行各种操作，包括：查找实体实例、持久实体实例、合并实体实例、删除实体实例、分离实体实例、刷新数据库、刷新实体实例、清空持久性上下文等。
- 涉及实体实例新建、更新和删除等的操作一般应该在事务内执行。本地资源型事务的控制在应用程序编程接口 EntityTransaction 中定义。
- Java 持久性查询语言(JPQL)用于定义基于实体的查询，以及批量更新和删除操作。JPQL 语句包括 SELECT 语句、UPDATE 语句和 DELETE 语句。
- JPQL 语句的执行需要持久性提供器支持，并通过查询 API 实现。查询对象可以通过调用 EntityManager 的 createQuery 等方法获得。
- 定义 JPQL 语句的方式有两种：一是所谓的动态查询，即在程序代码中创建 JPQL 语句字符串；二是所谓静态查询或命名查询，是指用@NamedQuery 标注定义 JPQL 语句并对其命名。

习　题　12

1. 什么是持久性单元？如何定义一个持久性单元？持久性单元的定义保存在哪个文件中？

2. 实体有哪几种状态？简述各状态的特点。

3. 简述实体管理器中有关实体操作的几种主要方法。

4. 何谓 JPQL？JPQL 包括哪些语句？如何执行 JPQL 语句？

5. 修改应用项目 sh6_lookandbuy(第 6 章习题 6)，使其从数据库 bookstore(第 11 章习题 4)中而非 model.DataBase 类中访问图书数据：

(1) 在项目中创建 bookstore 数据库的一个 JDBC 连接池，然后再基于该连接池创建一个 JDBC 资源。

(2) 基于上述 JDBC 资源，创建对应于数据库表 BOOK 的实体类 Book，并替换项目中原先的 Java 类 entity.Book。

(3) 基于上述 JDBC 资源，为项目定义一个持久性单元。

(4) 修改 model.BookManager 类中的相关方法，使其通过访问数据库完成相应的业务处理功能。

参 考 文 献

[1] GEARY D,CAY HORSTMANN. Core JavaServer Faces[M]. 3rd ed. [n. l.]:Prentice Hall,2010.

[2] GEARY D,CAY HORSTMANN. JavaServer Faces 核心编程[M]. 3 版. 王超,译. 北京：清华大学出版社,2011.

[3] MANN K. JSF 实战[M]. 铁手,程晓,何勇,译. 北京:人民邮电出版社,2007.

[4] BURNS E,KITAIN R. JavaServer™ Faces Specification Version 2. 0[OL]. http://www. jcp. org/en/jsr/summary?id=JSR+314.

[5] Oracle Corporation. The Java EE 6 Tutorial[OL]. http://www. oracle. com/technetwork/java/javaee/downloads/index. html.

[6] BURNS E. CHRIS SCHALK. JavaServer Faces 2. 0 安全参考手册[M]. 陶克,熊淑华,译. 北京：清华大学出版社,2012.

[7] KODALI R R,WETHERBEE J,ZADROZNY P. EJB 3 基础教程[M]. 马朝晖,杨艳,等译. 北京：人民邮电出版社,2008.

[8] PANDA D,RAHMAN R,LANE D. EJB 3 实战[M]. 马朝晖,等译. 北京:人民邮电出版社,2008.